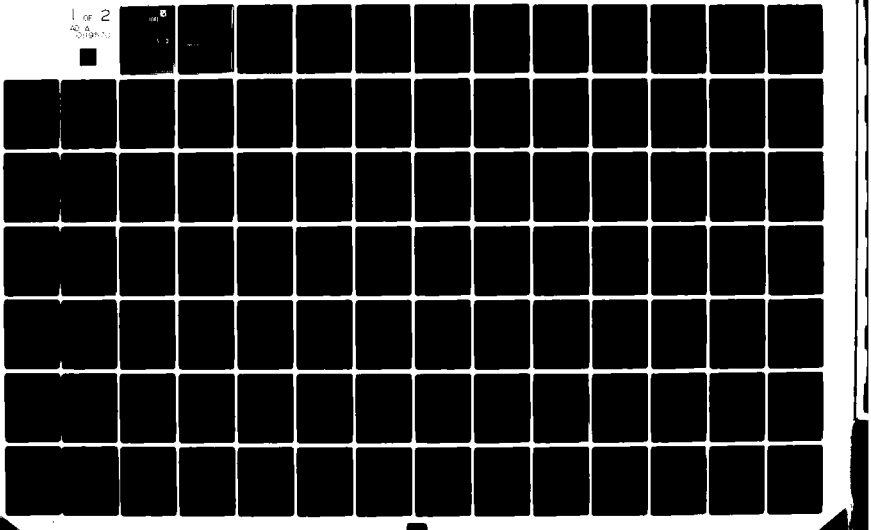
AD-A089 570 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/5
OPTIMIZED COMPUTER SYSTEMS FOR AVIONICS APPLICATIONS.(U)
FEB 80 R T CHIEN, L J PETERSON F33615-78-C-1559

UNCLASSIFIED AFAL-TR-79-1235 NL



AFAL-TR-79-1235

(12) BS

LEVEL II

OPTIMIZED COMPUTER SYSTEMS FOR AVIONICS APPLICATIONS

COORDINATED SCIENCE LABORATORY
UNIVERSITY OF ILLINOIS
URBANA, ILLINOIS 61801

February 1980

TECHNICAL REPORT AFAL-TR-79-1235
FINAL REPORT FOR PERIOD 1 FEBRUARY 1979 - 30 SEPTEMBER 1979

Approved for public release; distribution unlimited.

AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Dieter J. Schiller

DIETER J. SCHILLER
Project Engineer

David J. Brazil

DAVID J. BRAZIL, CAPT, USAF
Tech Mgr, Software & Processor Grp
System Technology Branch

FOR THE COMMANDER

Raymond E. Siferd

RAYMOND E. SIFERD, COL, USAF
Chief, System Avionics Division
Air Force Avionics Laboratory

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify [illegible] (AFWAL/[illegible]), WPAFB, OH 45433 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFAL-TR-79-1235	2. GOVT ACCESSION NO.: AD-A089570	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): OPTIMIZED COMPUTER SYSTEMS FOR AVIONICS APPLICATIONS.	5. TYPE OF REPORT & PERIOD COVERED: FINAL TECHNICAL REPORT. 1 FEB [illegible] - 30 SEP 1979,
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): R. T. CHIEN, L. J. PETERSON, et al.	8. CONTRACT OR GRANT NUMBER(s): F33615-78-C-1559
9. PERFORMING ORGANIZATION NAME AND ADDRESS: COORDINATED SCIENCE LABORATORY, UNIVERSITY OF ILLINOIS, URBANA, ILLINOIS 61801	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PE-61101F, AFAL 2003-03-41
11. CONTROLLING OFFICE NAME AND ADDRESS: AFWAL/AAAT, AF WRIGHT AERONAUTICAL LABORATORIES, AFSC, WRIGHT-PATTERSON AFB, OHIO 45433	12. REPORT DATE: Febr[illegible]980; 13. NUMBER OF PAGES: 184
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED; 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

SIGNAL PROCESSING, RADAR, ELECTRONIC WARFARE, COMMUNICATIONS, IMAGE PROCESSING, COMPUTER ARCHITECTURE

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The main purpose of this project is to investigate the commonalities among the four subareas of signal processing, namely, radar, communications, image processing and electronic warfare; and to establish possible common functional descriptions as the basis for a common architecture.

An extensive search was made to list all important kernels and algorithms in radar, communications, and image processing. (continued on reverse)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

011700

BLOCK #20 CONTINUED - ABSTRACT:

These kernels and algorithms were carefully analyzed with respect to their computational complexity and identification of commonality for architectural purposes. It was discovered that significant commonalities do exist in many areas. These common areas represent significant overlap and commonality which can be utilized in a common architecture.

FOREWORD

This final report covers work conducted by Coordinated Science Laboratory, University of Illinois. The work was performed under Contract F33615-78-C-1559, Project 2003, Task 03, Work Unit 41 for the U. S. Air Force Avionics Laboratory, Wright-Patterson AFB, Ohio 45433. The Air Force technical monitor was Dieter J. Schiller (AFAL/AAT-2).

The research covered by this report was conducted during the period 1 February 1979 to 30 September 1979. The report was released on 10 December 1979.

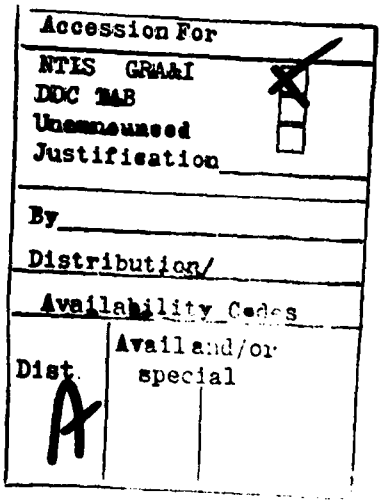


TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Cont'd)

SECTION I

INTRODUCTION AND SUMMARY

In February 1979, the Avionics Laboratory of the U.S. Air Force at the Wright-Patterson Air force Base contracted the Coordinated Science Laboratory, University of Illinois, to do a study in "Optimized Computer Systems for Avionics Applications." The main purpose of this project is to investigate the commonalities among the four subareas of signal processing, namely, radar, communications, image processing, and electronic warfare; and to establish possible common functional descriptions as the basis for a common architecture.

An extensive search was made to list all important kernels and algorithms in radar, communications, and image processing. These kernels and algorithms were carefully analyzed with respect to their computational complexity and identification of commonality for architectural purposes. It was discovered that significant commonalities do exist in many areas, namely:

data compression
convolutional transform
spectral transformation
linear filtering
vector operations
2D operations
table look up and entry
cross correlation

These common areas represent significant overlap and commonality which can be utilized in a common architecture. The details of these analyses are presented in Section II.

We recommend the following actions to follow up the promising avenues we have identified. Efforts should be made:

1. to carry out an in-depth study of current avionics digital signal processing;
2. to identify and characterize avionics signal processing kernels;
3. to investigate adaptive processing techniques;
4. to evaluate and analyze image processing techniques; and
5. to investigate common computer architectures for avionics signal processing.

Details of these tasks are given in Section II.

Other interesting work done on the project is reported in Sections III through VI. In Section III we present results on computer architecture for finite elements. In Section IV results are given on the evaluation of signal processing architectures. The results of a comparative study of avionics processors is given in Section V. Finally, the design of a hardware system for analyzing image processing kernels is given in Section VI.

SECTION II

INVESTIGATIONS INTO THE COMMONALITY OF SIGNAL PROCESSING FUNCTIONS IN RADAR, ELECTRONIC WARFARE, COMMUNICATION AND IMAGE PROCESSING SYSTEMS

During the last several years there has been a growing interest in the use of digital signal processing in airborne radar, communication and electronic warfare (EW) systems. More recently, interest has developed in the use of visual image processing systems in the avionics environment. Avionics systems are often centered around a general purpose computer that performs high level signal processing functions and coordinates the many components of the system. In an aircraft, numerous avionics systems are operating simultaneously and in close proximity. In the past, the trend has been to supply a specialized digital processor, complete with its unique software system, for each of these individual systems. This procedure has resulted in a proliferation of dedicated processors, accompanied by very high software and hardware development and maintenance costs. It appears that there would be a considerable cost savings if all the specialized processors could be replaced by one (or more) specially developed standardized processor that is capable of performing all the generic signal processing functions required in the avionics systems on board. The advantage of a common processor in multiple avionics systems on an aircraft results from the uniformity in both software and hardware development and maintenance costs. If several identical processors were present to accommodate peak processing loads, it is likely that overall reliability would be increased because of failure in one processor could be compensated by another functioning processor when all systems are not in simultaneous use.

2.1 Generic Signal Processing Functions for Radar, Electronic Warfare, Communication and Image Processing Systems

The objective of this research is to identify and characterize generic signal processing requirements that are common in radar, electronic warfare, communication and image processing systems. These results can be used in future work as a basis for developing a processor architecture that can operate efficiently in any one of the avionics signal processing environments.

2.1.1 Radar Systems

The fundamental structure of all radar systems is described by the block diagram of Figure 1 [1]. The matched filter, the detection processor and the post processor are usually custom built digital hardware units that are designed to process large quantities of data at very high rates. A typical sampling rate would be 25MHz. To achieve these high rates ECL circuitry is often required with an accompanying high power consumption. Fortunately, the wordlength is usually in the modest range of 8 to 12 bits, so in practice, the high sampling rate can be accommodated in the processors. Essentially these units act as preprocessors for the data processing computer, which maintains control of the entire system. It also implements high level signal processing functions that extract further information from the recorded data files. The high level functions include signature analysis, map matching, target tracking, data compression, or signal enhancement through digital processing techniques.

The digital matched filter is the heart of the entire system. It is a baseband correlator that correlates the returned signal with a stored replica of the transmitted waveform. Since it operates in the baseband with I and Q channels, it is most easily characterized as a complex, programmable, FIR

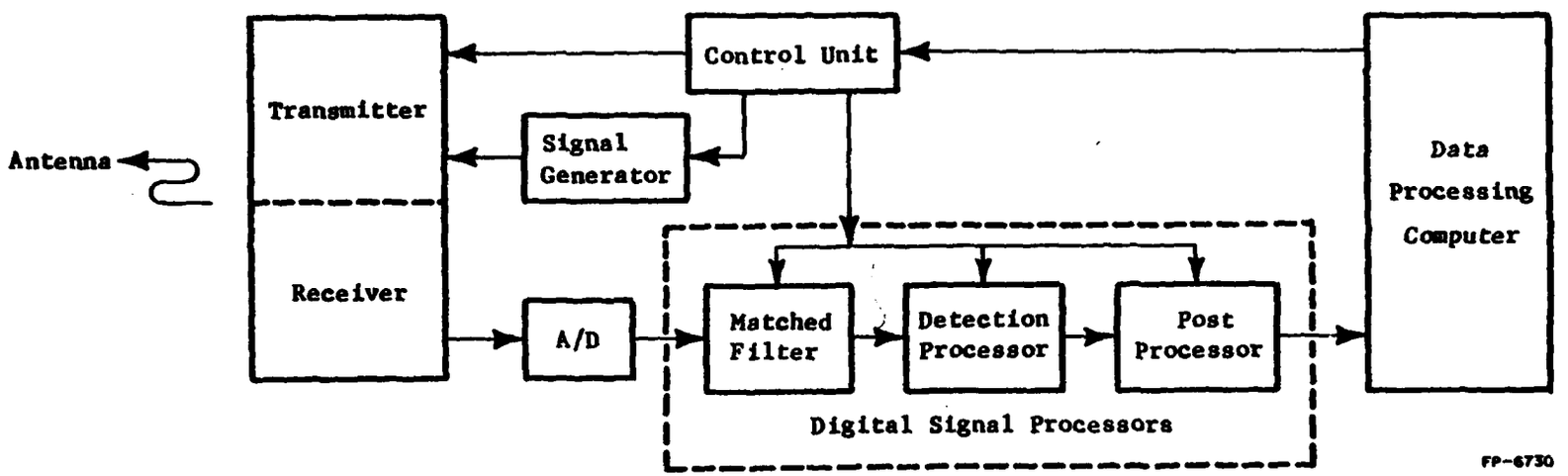


Figure 1. Fundamental components of a typical radar system.

filter described by Equation (1), where h(k) is the impulse response of the matched filter, x(n) is the input signal, and y(n) is the output signal from the matched filter.

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \qquad (1)$$

Normally the value of N is large (on the order of 1024) so that a direct computation of (1) is not possible at the required sampling rates. In this case the data is buffered and the finite convolution is implemented by fast Fourier transform (FFT) block processing. This approach brings into effect the $N \log_2 N$ computational savings of the FFT, which makes it possible to achieve the desired sampling rates. Since h(k) and x(n) are both complex baseband signals, the complex arithmetic of the FFT is very convenient for this problem. Since the filter function h(k) is programmable in a digital system, the central computer can make an on line selection of the specific waveforms that will be transmitted. The computer simply instructs the waveform control unit to load the matched filter with a replica so the proper correlation is performed.

An important class of airborne radars is referred to generically as synthetic aperture radar (SAR). SAR is a technique that is used to produce terrain images from data collected with a side-looking radar carried on an airplane or space vehicle. The coordinate perpendicular to the flight path (as projected on the earth's surface) is called the range. Resolution along the range dimension is controlled by conventional methods, i.e., by controlling the pulse length and by modulating the pulse with a two-sided linear FM reference waveform (chirping). The problem is that a satisfactory resolution along the flight path (azimuthal coordinate) cannot be obtained because a small physical antenna cannot provide the narrow beamwidth necessary

for good resolution along this dimension. The synthetic array principle overcomes this difficulty through signal processing techniques.

At regularly spaced positions along the flight path the radar transmits a coherent burst of radiation. The echo returns are demodulated into in-phase and quadrature (90° out of phase) video components, sampled with A/D converters if necessary, and then stored in a recording medium. The recording may be photographic (film) or electronic (tape, disk, or digital memory). After the recording has been completed at N distinct positions in space, the stored signals are combined by signal processing to create the equivalence of a long physical array that produces a narrow beam pattern. The synthetic array principle provides the desired azimuthal resolution at the cost of extensive signal processing.

An important class of digital processing algorithms for synthetic aperture radar (SAR) is based on the computational efficiency of the FFT algorithm. The combination of linear FM modulation and spectral analysis results in a processing algorithm that is sometimes called the "stretch" technique. The radar transmits a linear FM modulated pulse (chirped pulse). When the return signal is mixed with a linear FM reference waveform that has the same FM rate as the transmitted pulse, the difference frequencies that result from targets at different ranges will be distinct constant frequency components in the returns. If the spectrum of the mixed signal is computed, the targets at different ranges can be resolved as distinct spectral components. If the data is processed digitally, the spectrum can be computed with the FFT algorithm, resulting in a highly efficient processing technique.

When radar flies past a point target in the ground patch, the return from this point target at the N distinct pulse positions along the synthetic aperture represents samples of an FM waveform which is FM modulated by the

Doppler effect due to the relative motion between the radar and the target. Since the frequency modulation is approximately linear FM, there is a "natural stretch" in the recorded samples in the azimuthal coordinate. The operations of mixing with a linear FM reference waveform and computing the spectrum of the result can, therefore, also be used as the processing algorithm in the azimuthal coordinate system. Since a FFT must be computed in both the range and azimuthal coordinates, the processing is efficiently executed with a single 2D FFT algorithm. However, since the along track modulation due to the Doppler effect is only approximately linear FM,the 2D stretch technique is an approximate or "suboptimal" algorithm.

As a specific example consider the block diagrams of a synthetic aperture radar system shown in Figures 2 and 3. Figure 2 illustrates the analog functions that are normally implemented in the transmitter and receiver. The receiver output, consisting of baseband I and Q channels, is sampled and stored in memory at each position along the synthetic aperture. Figure 3 illustrates the digital functions that follow the receiver. The 2D interpolation, 2D FFT, and the amplitude detector are signal processors that are usually specialized hardware units. After the basic image is formed, the data is sent to the data processing computer for high level processing such as pattern recognition, image enhancement, or data compression. The final image is then displayed, recorded, or transmitted as required in the specific application.

2.1.2 Electronic Warfare Systems

The basic function of many EW systems is to collect inputs from many different RF sources, to analyze the signals in order to extract important parameters and classify the signals into an emitter file. It is important for

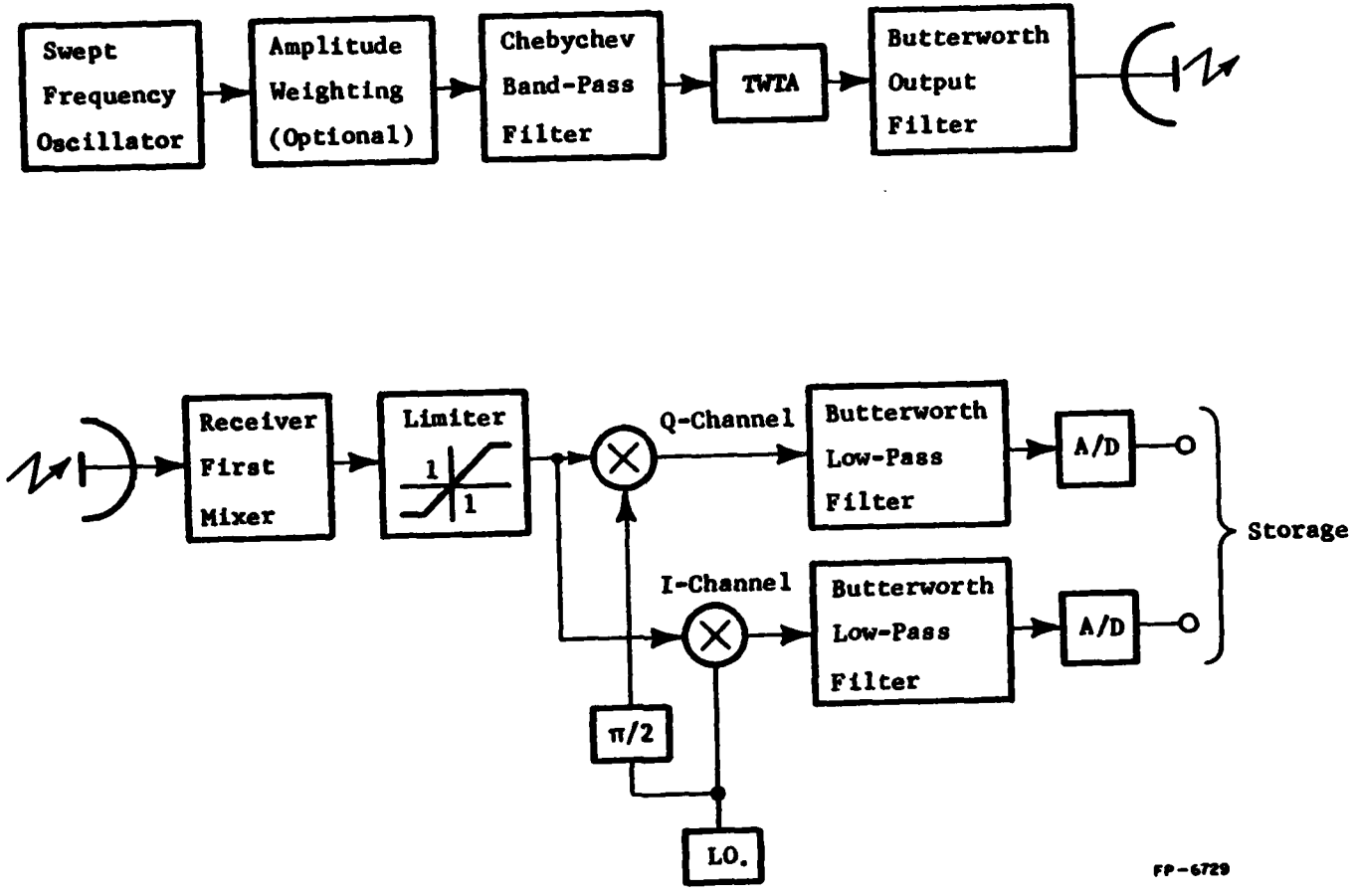


Figure 2. Transmitter-receiver functions in the front end of a SAR system.

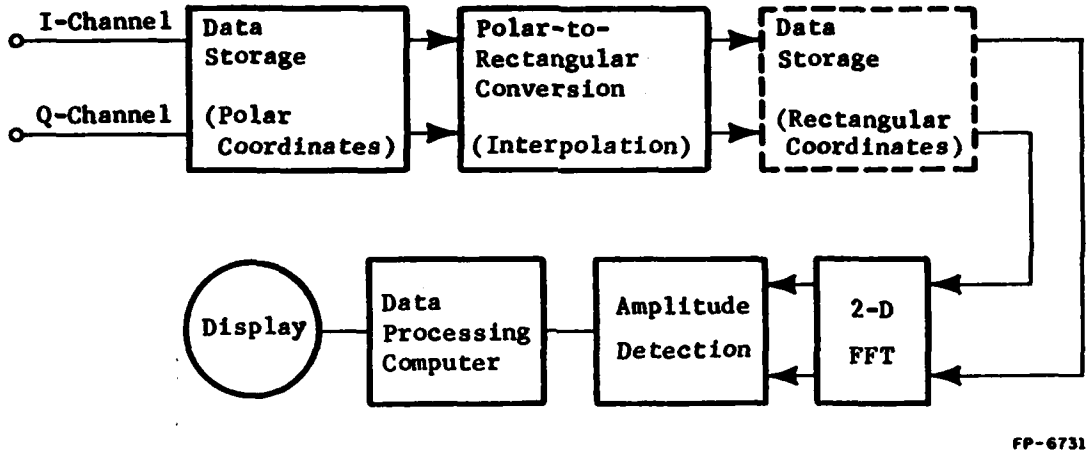


Figure 3. Block diagram of a spotlight mode SAR signal processing system.

the analyzing system to extract parameters that cannot be easily varied by the sender. For example, carrier frequency, pulse repetition rate and pulsewidth are easily changed. But the pulse shape is more difficult to change, since it is often controlled by the hardware parameters fixed in the transmitter.

A generic EW system is shown in Figure 4 [2]. The system consists of N receivers, each of which has its own antenna and can be tuned independently of the other receivers. The system discriminates among numerous sources and attempts to acquire data from each "emitter", which is then stored in a data bank and referred to as the emitter files. After demodulation, filtering and amplification the baseband signals are sent to a pulse characterization unit. The parameters extracted by the pulse characterization unit are used to excite a threat detector, which then updates the information in the emitter files and activates remaining portions of the EW system. The pulse characterization unit and the threat detector are under control of the central EW system computer. The central processor also controls an energy management subsystem that implements jamming if offensive response is needed.

An important computational task in the signal analysis processor is the calculation of error metrics for sorting and classifying incoming signals. Matched filtering, noise suppression filtering, spectral analysis and correlation also play important roles in preparing the raw data for error metric measurement. Three commonly used metrics are the least absolute, least squares and least infinite (Chebychev) metrics. These metrics are defined by the following equations, where $x_i(n)$ is referred to as the i-th template and $w(n)$ is a weighting function.

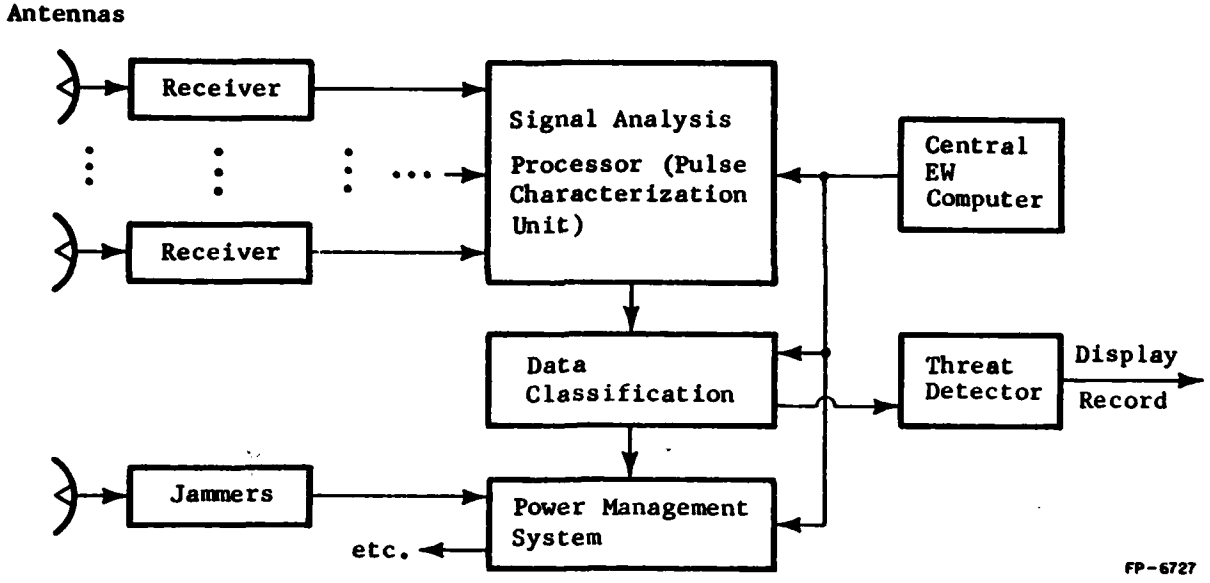


Figure 4. Block diagram of a typical electronic warfare system.

$$e_{LA}(i) = \sum_{n=0}^{N-1} w(n)|x(n)-x_i(n)| \quad \text{(Least absolute)}$$

$$e_{LS}(i) = [\sum_{n=0}^{N-1} w(n)[x(n)-x_i(n)]^2]^{1/2} \quad \text{(Least squares)}$$

$$e_{LI}(i) = \max_n [w(n)|x(n)-x_i(n)|] \quad \text{(Least infinite)}$$

The weighting sequence, w(n), is selected in such a way as to give certain data samples more or less influence on the value of the norm. For example, the window may be uniform ($w(n) = 1$, for $n = 0,\ldots,N-1$ and $w(n) = 0$ for n otherwise) so that all points in the observation window are given equal importance. Or the window may be exponential ($w(n) = \alpha^{N-n+1}$, for $n = 0,\ldots,N-1$ and $w(n) = 0$ for n otherwise) so that the data points that occur later in the observation window are given more influence in the value of the norm. The choice of metric depends on the characteristics of the noise on the incoming signals. The least infinite metric is best for uniform noise, while the least absolute metric is best if the noise distribution is characterized by long tails. For example, the least infinite metric is often used when the noise is Gaussian white noise whose power spectral density is uniformly spread across the bandwidth of the system. The least squares metric is appropriate for narrow band colored noise whose energy is highly concentrated around preferred frequencies and tapers off rapidly at other frequencies. These metrics can be characterized mathematically as p-norms that are familiar from the theory of normed vector spaces.

The major function of the pulse characterization unit is to determine the pulse shape. This can be done by detecting pulse amplitude, pulsewidth, rise time, and fall time. Rise time and fall time can be estimated by digital differentiators constructed as linear phase finite impulse response digital filters. Alternately, the spectrum could be computed by means of a FFT

hardware unit in order to determine the spectral content of the pulse. Since the spectral content and the pulse shape have a unique correspondence, FFT analysis is a viable pulse characterization technique if high speed hardware FFT units are available. Other parameters, such as time of arrival, are not effective for presorting and classification, but they may be important in threat analysis. Careful analysis of pulse strength and Doppler history may indicate the approach of hostile transmitters.

2.1.3 Communication Systems

In communication systems there are many different functions that might be implemented digitally. These include digital filtering, mixing, multiplexing, error correction coding, and data compression (for both image and speech signals). In general the RF sections of receivers and transmitters must be analog because digital hardware cannot operate at high enough sample rates to accomodate RF frequencies. After the signals have been translated to IF frequencies or to baseband frequencies by means of analog mixers, the signals can then be converted to digital form and processed digitally from that point on.

Data compression is one of the important functions that might be used to reduce bandwidth requirements on a communication channel. Data compression techniques can be classified into time domain and transform domain compression methods. Differential pulse code modulation (DPCM) is a time domain compression scheme that is effective for transmitting both image and voice signals. The basic elements of the system are shown in Figure 5. Both the compression and decompression functions require digitally implemented linear predictors that function to remove the correlated signal components in the data [3]. This results in a transmission of the uncorrelated differences that

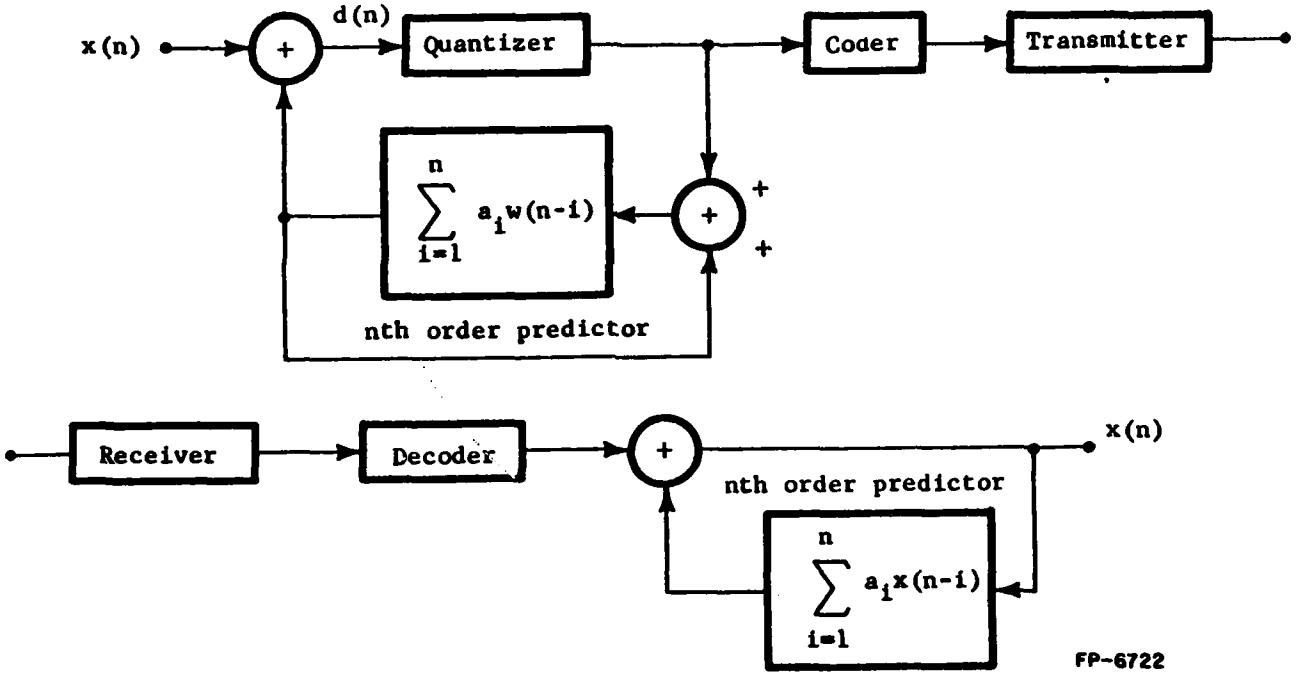


Figure 5. Data compression in communication channels with DPCM.

occupy a significantly compressed dynamic range. The signal processing at each end of the channel requires an efficient implementation of a N-th order digital filter. It is found in practice that N=3 is normally adequate in this application. If the data can be accurately modeled as a stationary stochastic process, the optimal filter coefficients (a_i's) can be calculated offline and the filtering can be accomplished with time invariant filters. DPCM has been used successfully in time domain compression of images, although the quality of the reconstruction is highly dependent on the exact nature of the image. Recently, Motorola Inc. has been investigating the use of DPCM for lowering the bandwidth requirements in hand-held two-way radios.

Transform domain compression techniques are among the best known data compression techniques because of their widespread usage in image compression for the space programs of the 1960's. One way to envision these techniques is to describe them as a coordinate axis transformation which translates the data into a domain where the signal components become decorrelated. Another popular intepretation is that the transformation is a mechanism for localizing the energy content of the signal, so that only the significant components are selected for transmission. The discrete Fourier transform (DFT) is one effective compression technique.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}nk} \qquad k=0,\ldots,N-1.$$

The N values of X(k) are usually computed by means of an FFT algorithm. A typical image will have most of the energy concentrated in the low frequency portion of the spectrum. Therefore, only a subset of the X(k)'s need to be transmitted. At the receiving end of the channel, the inverse DFT is calculated,

$$\hat{x}(n) = \frac{1}{N}\sum_{k=0}^{N-1} \hat{X}_{(n)} e^{j\frac{2\pi}{N}nk} \qquad n=0,\ldots,N-1.$$

where $\hat{X}_{(k)}$ is an approximation to X(k) obtained by inserting zeros into the positions that correspond to components that were not transmitted.

The Walsh-Hadamard transform (WHT) is another well known data compression algorithm. It functions in much the same way as the DFT described above, although it is a very computationally efficient transform because the weighting coefficients in the Hadamard matrix contain only 1 and -1 values. The Hadamard matrix is generated recursively from "direct matrix products,"

$$\hat{H}_N = \hat{H}_2 \times \hat{H}_{N/2}, \qquad N = 4, 8, 16,\ldots$$

where $$\hat{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

For example, $$\hat{H}_4 = \hat{H}_2 \times \hat{H}_2 = \begin{bmatrix} \hat{H}_2 & \hat{H}_2 \\ \hat{H}_2 & -\hat{H}_2 \end{bmatrix}.$$

The WHT has been used in image processing primarily for 1) feature extraction, 2) bandwidth (data) reduction, and 3) dimensionality reduction.

Both the DFT and the WHT, when applied to 2D data processing, are members of a class of linear operators called unitary transforms. For most kinds of processing, the well known Karhunen-Louve transform is optimal because it results in a set of data in a coordinate system where the components are uncorrelated. Unfortunately, it is computationally unviable, requiring recomputation of the transform matrix for each new set of data. Also it does not possess a fast algorithm. In practice the FFT and the WHT are widely used because they are near optimal transforms that are computationally attractive.

Two areas of extreme importance for digital communication systems are error-correction coding and spread spectrum modulation. In essence, these two areas are closely related through the use of specially designed PN codes in both direct sequencing and frequency hopped spread spectrum systems. Error correcting encoders and decoders are characterized as computationally simple devices that consist largely of logic gates and shift registers. These devices are usually (but not always) located toward the front end of the system, where they are required to operate at very high real-time data rates. Therefore, for the most part, encoding and decoding is not implemented in programmed signal processors, but rather is implemented by high speed custom made circuits that are becoming more and more highly integrated. In some systems these high speed circuits may be under the control of a microprocessor, although the microprocessor is too slow to do the actual processing. The situation is much the same in direct sequenced spread spectrum systems, where primary signal processing involved in despreading is a high speed correlation of the incoming data with the known PN modulation function. Once phase lock is obtained, the correlation behaves essentially as a binary demodulator. Once these issues are understood, it becomes apparent that the functions of error-correction and spread spectrum modulation do not, in general, belong in the system block that we have previously identified as the digital signal processor or the data processing computer. Rather they should be considered as integral parts of the transmitter and receiver that are best implemented in LSI technology.

2.1.4 Image Processing Operations

Standard image processing systems input TV images, digitize the images, and store the images in arrays, typically of 250 rows, 250 columns and with six to eight bits of intensity for each point. In real time with a raster

rate of 30 frames per second, this is a significant number of bits per second on which to do detailed processing. Image processing systems therefore are designed to separate the important, significant features of an image from its random or insignificant features. The ideal image processing system would describe the images in terms of objects and relationships between objects, and in terms of the relationships between objects and the viewer (e.g., position or location of objects, identity of type of object, velocity, size, etc.).

In order to determine a set of kernels for image processing operations, a study was made of the functions developed for image processing tasks. At this time it is not known what specific tasks would be required of an image processing system in the avionics environment. It is anticipated, however, that such tasks as landmark and target recognition, target tracking, and map matching would be among the set of avionics image processing tasks. The functions described in this section include many of those which are suitable for these tasks.

2.1.4.1 Edge Detection Algorithms

Edge detection algorithms operate directly on the stored image array and generally attempt to find significant differences between neighboring picture elements (pixels). If enough of these differences occur locally, the interpretation is that there is a local edge element in that part of the image array.

The Roberts Cross Operator

The simplest of the gradient operators is the Roberts cross operator [4]. The gradient at each point is approximated from intensity levels at four pixels in a 2 by 2 square centered at that point. The difference between the grayscale values at the upper left and lower right of the square is an estimate of the directional derivative along a line turned 45 degrees clockwise from the horizontal. Likewise, the difference between the upper right and lower left is an estimate of the directional derivative along a line perpendicular to the first line. The square root of the sum of the squares of these two differences is an estimate of the steepest gradient at that point, regardless of its direction. In this computation, both differences that are calculated are convolutions of the image array with the following two matrices:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

The most striking feature of this operator is its extreme simplicity. It uses the grayscale values over a window containing only four pixels. Once these have been fetched, two subtractions, two multiplications, an addition and a square root are performed. If the gradient is being thresholded, then the square root operation can be eliminated by using the square of the threshold. Another application which eliminates the squaring operation simply sums the absolute values of the results of the two convolution operations and then thresholds the result. The simplicity of the cross operator is offset by its extreme susceptibility to noise in the image. The small area used enables a single noisy point to drastically change the gradient estimate at that point resulting in a noisy gradient picture. Hence, the use of this operator is

limited to fairly clean images.

The basic operations used with this operator are convolution and a norm operation.

The Sobel Gradient Estimator

Sobel's [5] method for finding the gradient uses the following two convolutions to find the x and y directional derivatives, respectively:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The approximate gradient is then obtained by taking the square root of the sum the squares of the x and y directional derivatives.

This gradient estimator has essentially the same advantages and disadvantages as the Roberts cross operator. Namely, the number of arithmetic operations to be performed is small, but owing to the small size of the window, a noisy image results in a noisy estimate of the gradient. Note that the vector of weights (1,2,1) used in the two convolution masks need only be applied to each row and each column of three pixels only once over the whole image. These results can then be used to evaluate the x and y directional derivatives more efficiently. In particular, each application of Sobel's operator can be performed with seven additions, two subtractions, two multiplications, and one square root, on the average.

The basic operations used with this operator are convolution and a norm operation.

The Burr Edge Detection and Following Operator

The Burr method performs gradient detection and edge tracking over a gray-scale image in a single application [6]. It does not require intermediate storage and subsequent thinning as in most two pass procedures. (edge detection followed by edge correlation) Each pixel in the input image is examined in a raster-scan fashion. There, a gradient and its orientation is computed based on orthogonal differences:

$$DX = [F(i,j+1)+F(i,j+2)+F(i,j+3)] - [F(i,j-3)+F(i,j-2)+F(i,j-1)]$$
$$DY = [F(i+1,j)+F(i+2,j)+F(i+3,j)] - [F(i-1,j)+F(i-2,j)+F(i-3,j)]$$

$$\text{Gradient} = (DX^2 + DY^2)^{1/2}$$
$$\text{Edge Angle} = \arctan(DY/DX)$$

Local edge maxima are found by parabolic interpolation on three successive gradient values using the following formula:

$$(g_3 - g_1)/(4g_2 - 2g_1 - 2g_3).$$

where the g_i are the gradient values of three points in either the x or the x direction. The coordinate of the maximum value is then found and compared with the end point coordinates. If it falls between the end points, it is chosen as a potential edge point. If no intermediate maximum exists, no edge point is flagged. Note that the calculation of the discrete differences is effectively an averaging operation and smoothes the effects of spikes. If the gradient exceeds some threshold and it is at a local maxima in the X or Y direction, that pixel is termed an edge point. Discovery of edge points

initiates examination of nearest-neighborhood pixels for more edge points. The searching terminates when no further edge candidates are found beyond three neighboring pixels. The edge points' gradients and positions are stored in the sequence they are found.

The algorithm constrains the direction in which to search for edge points in the direction of the last edge found according to the computed gradient angle at that point. Only neighborhood pixels within a fan beam in this direction are examined. The beam width is chosen large enough to maintain continuity of edge following in noisy images.

From a computational standpoint, this edge finder has the desirable feature that no area convolutions are required, thus avoiding double indexing. Instead, both the x and y gradients are evaluated using points along a single row or column of pixels. In the most efficient implementation (applying the operator over the whole image), an average of ten arithmetic operations are required per pixel plus about twenty more for every one that becomes an edge point.

The basic operations used include table entry, thresholding, convolution, norm and interpolation.

Compass Gradient Operators

A two-dimensional discrete differentiation can be performed by the convolution of a 3 by 3 Compass Gradient mask over an image [7,8]. A new pixel is computed as a function of pixels in a 3 by 3 neighborhood centered about the pixel under consideration. The function is linear since the masks contain integer elements. They are chosen such that their sum is typically 0 or 1, to prevent overly large gradients from being computed. The 3 by 3 masks

given below give maximum response in eight principal (compass) directions:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & -1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & 1 \\ -1 & -2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & -1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & -1 \\ 1 & -1 & -1 \end{bmatrix}$$

NORTH | NORTH-EAST | EAST | SOUTH-EAST | SOUTH | SOUTH-WEST | WEST | NORTH-WEST

A similar set of masks known as 5-level simple masks more closely approximates the partial derivatives in each of the eight directions. (see below)

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

NORTH | NORTH-WEST | WEST | SOUTH-WEST | SOUTH | SOUTH-EAST | EAST | NORTH-EAST

Here, zero weighting at the center of the masks suppresses jitter along the line near the occurrence of an edge. Computation requires only eight multiplications and seven additions per pixel during the convolution. Usually an edge direction map is also desired, in which case all eight directional gradients are computed. A computational advantage is that only four out of the eight masks need be computed. This is because the masks for opposite directions are symmetric, and thus their gradients are opposite in sign. So computing, say mask EAST, one can decide on an edge direction of EAST or WEST, depending on whether the gradient is positive or negative. Another advantage of 5-level directional masks is a characteristically higher gradient amplitude in the diagonal directions.

The Kirsch operator provides a further example of the use of a 5-level convolution mask [9]. All eight gradients are computed for a pixel neighborhood and the gradient of highest absolute value is chosen as the gradient at that point. Shown below are the compass masks for this operator.

$$\begin{bmatrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix} \begin{bmatrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{bmatrix} \begin{bmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{bmatrix} \begin{bmatrix} -3 & -3 & -3 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{bmatrix} \begin{bmatrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{bmatrix} \begin{bmatrix} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{bmatrix} \begin{bmatrix} 5 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{bmatrix} \begin{bmatrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}$$

NORTH | NORTH-EAST | EAST | SOUTH-EAST | SOUTH | SOUTH-WEST | WEST | NORTH-WEST

If the chosen gradient is greater than some threshold, the gradient and its orientation are saved in the gradient map. Thus a two-dimensional feature vector can be utilized, which leads to more flexibility and reliability. For example, edge linking using the orientation parameter can reinforce an edge candidate based on the gradient. The algorithm requires eight times the computation of a convolution using a 3 by 3 operator.

The basic operations used are convolution and thresholding.

The Wallis Operator

Based on homomorphic image processing, this method attempts to solve the problem of detecting edges in different regions of illumination [10]. The relative magnitude between the logarithm of a pixel's luminance and that of the average of its four nearest neighbors (above, below, left, right) is computed. An edge is said to exist if it is greater than some threshold.

$$G(j,k) = \log[F(j,k)] - .25*\log[F(j,k-1)] - .25*\log[F(j,k+1)] - .25*\log[F(j-1,k)] - .25*\log[F(j+1,k)]$$

$$= .25*\log[F(j,k)^4/(F(j,k-1)*F(j,k+1)*F(j-1,k)*F(j+1,k))]$$

Here F(j,k) is the value of the pixel in position (j,k) of the image array.

This operation is essentially the application of a Laplacian operator to the logarithms of the pixel values. About the same amount of computation is required for this algorithm (8 multiplications + 1 division per pixel) and a 3 by 3 linear operator convolution. (8 multiplications + 7 additions per pixel). In performance, it picks up detail in both shadowed and illuminated regions. However, the edges are grainy and appear weaker due to the Laplacian operation.

The basic operations of the algorithm are the logarithm operation, convolution, and thresholding.

The Yakimovsky Edge Finder

In an attempt to find boundaries between areas of not only differing brightness but also differing texture, the Yakimovsky algorithm computes both the mean and the variance of adjacent neighborhoods [11]. In order to obtain a statistically significant estimation of these parameters, neighborhoods of suitable sizes are used. Among the shapes of the neighborhood pairs used are the following which are applied in the x direction:

```
 LLLL  RRRR          L  R
LLLLLLRRRRRR        LL  RR
 LLLL  RRRR        LLLRRR
                   LL  RR
                    L  R
```

where L stands for points in the left neighborhood, R for points in the right

neighborhood. These neighborhood shapes are rotated for application in the y direction. A maximum liklihood estimate is computed from the standard deviations of the two neighborhoods and from the combined neighborhood. This estimate is then compared against a threshold value for determining whether an edge exists between the neighborhood pair.

The basic operations used in this algorithm include histogram computation, mean computation, probability computation, and thresholding.

The Hueckel Operator

The Hueckel operator identifies edge fragments in a digitized image [12,13]. It examines a sequence of disks of pixels in the image array and returns the parameters c,s,p,d,b which describe the best fit line in each disk. The parameters c,s,p describe the line through the disk which separates the region into two areas of greatest intensity difference. The line defines a step function, F, which has intensity b on one side of the line and b+d on the other side of the line according to the following relation.

$$F(x,y,c,s,p,d,b) = b \qquad \text{if } cx + sy \leq p$$
$$F(x,y,c,s,p,d,b) = b + d \qquad \text{if } cx + sy > p$$

The parameters contain the slope information and step size of the line. In addition the confidence of the line parameters can also be ascertained. The Hueckel technique is widely used in line following strategies because it returns line direction information. The large disks which are used in the Hueckel operators make them relatively insensitive to noise in the image. This makes them useful on poorly quantized images, like the 16 level pictures that Hueckel used in his original work.

The Hueckel operator consist of linear and a nonlinear part. In the linear part a disk shaped window W(x,y) in the image E(i,j) is multiplied by a set of eight discrete mask functions H(p,x,y). For a window centered at i,j the multiplication is defined by:

$$a(n) = \sum_{W} H(n,x,y)E(x+i,y+j) \qquad n = 0,1,2...7$$

The disks used by the first Hueckel algorithm is quite large. The areas of reasonable disks are 32, 52, 69, 88, and 137 pixels on a square grid. The mask functions are the first eight functions of an orthonormal basis which spans the space of all image functions in the window W(x,y). The masks are weighted by the function w(x,y):

$$w(x,y) = \text{sqrt}(1 - r^2) \qquad \text{for } 0 \leq r \leq 1$$
$$w(x,y) = 0 \qquad \text{for } 1 < r$$

$$\text{where } r^2 = x^2 + y^2$$

This function gives less weight to pixels on the perimeter of the disk W(x,y) and thus reduces edge aliasing effects. The mask functions used in the algorithm are given by:

$$H(0,x,y) = c(0) * w(x,y) * (2r + 1)$$
$$H(1,x,y) = c(1) * w(x,y) * (5r - 2)$$
$$H(2,x,y) = c(2) * w(x,y) * 3x$$
$$H(3,x,y) = c(3) * w(x,y) * 3y$$
$$H(4,x,y) = c(4) * w(x,y) * (x^2 - y^2)$$
$$H(5,x,y) = c(5) * w(x,y) * 2xy$$
$$H(6,x,y) = c(6) * w(x,y) * 4x * (2r^2 - 1)$$
$$H(7,x,y) = c(7) * w(x,y) * 4y * (2r^2 - 1)$$

where c(0) through c(7) are constants of normality.

The nonlinear part of the Hueckel operator determines the line parameters <c,s,p,d,b> from the eight coefficients a(n). First the extrema of A(c,s) must be determined.

$$e_0(c,s) = a(2)c + a(3)s$$
$$e_1(c,s) = a(4)c + a(5)s$$
$$e_2(c,s) = a(1) + a(6)(c^2 - s^2) + a(7)2sc$$

$$A(c,s) = e_0 + \text{sign}(e_0) * \text{sqrt}(e_1^2 + e_2^2)$$

For the c,s which are extrema of A(c,s), and

$$p = e_2/1.41*(e_1 + \text{sign}(e_0) * (e_1^2 + e_2^2))$$

$$d = 4A/3.07*(1 - p^2)^2(1 + 2p^2)$$

$$b = a(0) - d(4 + p(3 + p(2 + p)))(1 - p)^2/8$$

From the c,s,p,d,b which are determined the confidence of the line is found, and if it is within bounds then the line parameters c,s,p,d,b are returned by the operator.

An extension of this operator provides an enhanced version of the above algorithm. In this algorithm, lines are now specified by the seven parameters $\langle c,s,p_1,p_2,d_1,d_2,b\rangle$.

$$F(x,y,c,s,p_1,p_2,d_1,d_2,b) = b \qquad \text{if} \quad cx + sy \leq p_1$$
$$F(x,y,c,s,p_1,p_2,d_1,d_2,b) = b + d_1 \qquad \text{if } p_1 < cx + sy \leq p_2$$
$$F(x,y,c,s,p_1,p_2,d_1,d_2,b) = b + d_2 \qquad \text{if } p_2 < cx + sy$$

This operator also consists of a linear and a nonlinear part. The linear part is almost identical to the above algorithm except that nine mask functions are used instead of eight. They have the following definition:

$$a(n) = \sum_W H(n,x,y)E(x+i,y+j) \quad n = 0,1,2...8$$

where

$$H(0,x,y) = c(0) * w(x,y) * (3r^2 - r + 1)$$
$$H(8,x,y) = c(8) * w(x,y) * (-21r^2 + 17r - 2)$$

and

H(1,x,y) through H(7,x,y) are the same as before

In addition to being able to recognize a wider class of lines the extended Hueckel operator also has all the advantages of its predecessor, namely, good noise immunity, line direction information, and a line confidence measure.

The basic operations in this algorithm include convolution, thresholding and norm operations.

2.1.4.2 Edge Following Algorithms

The edge following algorithms are ordinarily applied to edge point arrays created by the edge detection algorithms. Edge followers link together neighboring edges as part of the process of locating the boundaries of objects.

Edge Linking Using Position/Orientation Constraints

This algorithm is used to find groups of edges that link into approximate straight lines [14]. It achieves this in highly textured scenes and without *a priori* knowledge about the objects in the scene. The procedure is as follows:

1. Edge elements produced by the edge detector are given an x,y position and an orientation angle.
2. The 360 degree direction range is divided into equiangular intervals. For each interval, a list of edges is made whose edge orientations are within a given angular tolerance of the interval's mean angle.
3. For each interval, the coordinate system is rotated so that the new x-axis is along the direction of the interval's mean angle.
4. The plane is divided into vertical strips or buckets, typically three pixels wide. A list of edges is made for each bucket, sorted by y-coordinate.

5. Edges within a bucket are linked if their separation is less than a given threshold TX.

6. Edges in different buckets are linked if their orientations agree within a given tolerance, and if their x,y separations are below thresholds TX,TY.

7. Only those resulting segments with more than a specified number of linked edges are retained.

The proximity thresholds TX,TY are a function of the geometry of the initial edge detecting scheme and also of the bucket resolution. They can be further increased to link distant edges. From this, it might be expected that the algorithm would be very sensitive to slight variations in threshold selection. Experiments indicate that it is able to handle a wide variety of images, and can be used in a general purpose boundary detection system.

The computation involves processing of isolated edges or feature comparison with its neighbors. The computing costs are proportional to the number of edges processed. With the small number of bucket edges to be sorted and their raster ordering, the sorting computation is proportional to the number of pixels.

The basic operations used are convolution (spatial transformation), table lookup (or linked lists), and thresholding.

Bug Following Algorithm

This algorithm locates boundaries of objects in binary images. The procedure is sequential and utilizes a single demon ("bug") to track along a region's boundary [10]. The path taken is the desired boundary map. First, a candidate boundary point is located and the bug is started in one of its neighborhood pixels. It moves in the direction toward the located boundary point and into the adjoining pixel. The bug makes a left-hand turn if the

crossing lands the bug into an interior boundary point (black pixel). If white, (boundary exterior), a right-hand turn is made. This causes the bug to move in an oscillatory fashion along the boundary edge. In both cases, the x,y location of the crossings are saved. The algorithm terminates when the initial point is reached. The boundary crossing points form the boundary contour.

However, there are difficulties with such a simple algorithm. Regions with "spur" pixels (a single pixel which is diagonally adjacent to the region's interior) are sometimes missed. This is a consequence of the dependency of the resultant boundary map on the initial starting point. Also, in noisier images, the bug can easily become lost. A provision to allow backtracking and memory of preceding steps can overcome this problem. In addition, the algorithm can be extended to gray scale images in which boundary crossings are defined as sufficiently large differences in luminance between neighboring pixels.

The computation required for this algorithm is a function of the boundary's length. The algorithm is fast due to relatively small storage requirements and a dependency on logical operations.

The basic operations involved are indexing and thresholding.

Boundary Tracking from a Gradient Map

This algorithm is a sequential, demon-like procedure which is applied to a non-thresholded gradient map [15]. The pixel with the largest gradient in the image is chosen as the initial edge point. In its 3 by 3 neighborhood, another edge point with the highest gradient value is picked. If two or more pixels have the same maximum gradients, then an arbitrary choice is made. The

pixel with the maximum gradient in the neighborhood of the initial point is chosen as the second point. Then the following process is iterated. In its 3 by 3 neighborhood, the three pixels "closest" to a line formed by the previous two edge points are inspected. The one with the highest gradient is chosen as the next boundary point. Again, if two adjoining pixels have the same maximum, an arbitrary choice is made. If they are adjacent, the pixel in line with the previous points is chosen. This last constraint biases the following to locate straight lines. As edge points are found they are added to a list.

The procedure works well for noise-free images. However, noise usually sends the tracking way off the boundary. This can be alleviated by smoothing the input image before tracking. A second alternative is to use average gradient values. Average gradients are computed over an angular region about the line along the previous 2 edge points. The new direction is the direction of the largest gradient computed. The size of the bug increases and effectively implements a smoothing operation. In general, either method is useful in only low-noise images.

The 3 by 3 neighborhood bug algorithm does not require any arithmetic computation. Its operations are limited to finding maxima and to storing edge locations. The averaging enhancement does require the computation of averages.

The operations used here are thresholding, indexing and averaging.

2.1.4.3 Curve Fitting Algorithms

The curve fitting algorithms attempt to find concise representations of the edge string information, found by curve following algorithms, using various curve fitting algorithms. The end result is a significant compression

of the original data.

Line-Fitting by Ramer's Method

Ramer [16] developed the following technique for approximating a contour with a piecewise continuous chain of line segments. First, the endpoints of the contour are joined by a single line segment. Next, that point on the contour which is most distant from the line is found. If the distance is less than a specified value, then the line fit is satisfactory; otherwise, the contour is approximated by two line segments from the farthest deviation to the original end points of the contour. The routine is then called recursively on each of the two new segments until a sequence of line segments is generated which are everywhere within the specified distance from the data points.

The algorithm calculates the parameters of the line passing through the beginning and end points of the section of curve being approximated. Then for each point of the curve, the y distance to the approximating line is computed. This distance is simply the absolute difference between the value of y of the line computed from the x value of the point on the curve, and the y value of the point on the curve. The maximum y difference, y_m, yields the maximum distance, d_m between the curve section and the approximating line from:

$$d_m = y_m \cos\theta$$

where θ is the angle of the approximating line. for θ larger than 45°, the difference in x is computed to yield d_m. The value of d_m is compared with a threshold, and if it is less than the threshold, the line segment is used to approximate the section of curve. Otherwise the process is repeated on the

new line segments obtained from the point at maximum distance.

The computation involved includes a simple slope and intercept calculation for each approximating line, and a simple difference for each point on the curve segment. The maximum value of y is found by comparison of these differences, and the maximum distance between the curve and the line is calculate using a cosine function which is computed using a norm function.

The basic operators used are table entry, table lookup, the norm operator and simple distance computation.

The Hough Line-Fitting Method

The Hough transform maps edge points in cartesean space into ρ-θ space by the mapping [5]:

$$\rho = x_i \cos\theta + y_i \sin\theta$$

This mapping maps each (x_i, y_i) onto a sinusoidal curve in ρ-θ space. Through this mapping every colinear point in cartesean space is mapped onto a single point in ρ-θ space. This single point is the intersection of all the curves corresponding to the points on that line. The process is done digitally by forming a two dimensional histogram whose coordinates are discrete values of and θ. For each point (x_i, y_i) θ is varied full scale and the corresponding bins for ρ are incremented. When all the points have been transformed, the histogram is analyzed. Those ρ-θ pairs with more entries than a specified threshold are tagged as corresponding to lines in the original image.

Since any line can be represented as:

$$y = ax + b,$$

the parameters (a,b) can be determined directly from ρ-θ values from the relations:

$$a = \tan(\theta - 90^{\circ}) \quad \text{and} \quad b = \rho/\sin\theta.$$

One problem this algorithm has is that it does not distinguish colinear line segments. Instead it finds a single line through all colinear line segments. In application, additional heuristics are added to the algorithm to tag the line segments.

The basic operations used are table lookup for the sin, cos and tan functions, histogramming and thresholding.

Contour Approximation by Circular Arcs

Mckee and Aggarwal [17] approximate long digitized contours by a sequence of circular arcs. The points on the contour are plotted on a graph of tangent angle versus arc length, where the tangent angle is found from the line segments connecting successive points along the contour, and arc length is the summation of the lengths of these line segments. Because of the grid structure of the digitized images, and thus of the contour, the tangent angles are quantized to multiples of 45 degrees, and the ensuing plot is smoothed. This is accomplished by replacing each data point with the average of itself and four data points on either side. If the plotted tangent angle varies linearly with the arc length, then the contour must be following a circular

trajectory. Hence, circular arcs in the contour can be found by locating straight lines in the plot of tangent angles versus arc length.

The problem is thus reduced to that of fitting line segments to a one-dimensional waveform, typically by the method of least squares. The slope and length of each fitted line segment, which correspond to the curvature and the length of the arcs in the original contour, are then used as the final representation for the contour.

The basic operations used in this algorithm include table lookup and table entry, least squares computation and mean value computation.

2.1.4.4 Region Growing Algorithms

Region growing algorithms attempt to segment the image array by applying a similarity measure locally to the pixels. If a given pixel is similar to the second, it is added to the region list of the second pixel. More global region growing is done through a series of region merging and splitting operations which depend on other criteria.

Ohlander's Method for Image Segmentation

Through a series of linear transformations on the red, blue and green images of digitized color images, Ohlander's method produces six additional images corresponding to hue, saturation, grey level intensity, and the television industry standard parameters "Y", "I", and "Q" [18]. Regions are formed by generating a histogram of intensity values for each of the nine images, and wherever a pronounced narrow peak occurs in one of the images, it is supposed that all pixels contributing to that peak belonged to the same region. One advantage of this method is that all pixels belonging to a large

region of the picture but split into several parts by occluding objects can still be identified as a single region. Successive histogram partitioning on fragments of the whole picture could yield successively smaller regions when necessary.

The main kernel used for this method of region growing is that for compiling histograms, as it is utilized at least nine times per scene. In addition to this the basic operations are the computation of vector inner products in producing the transformations, and the thresholding operation.

A Region Merging Algorithm

One method for partitioning an image into regions is to initially assume that each pixel is an isolated region. Then, in successive stages, regions can be merged forming larger and larger regions until all adjacent regions are sufficiently different that no further merging is needed. An algorithm which was developed at the Coordinated Science Laboratory efficiently determines which pair of adjacent regions among all pairs are most nearly alike and merges them.

Before region merging in the above manner can be carried out, the number of regions must first be reduced from 60000, the number of pixels in the image, to about 1000. Hence, a first pass of non-optimal region merging is employed by a simple scanning operation. During this initial phase, every pair of adjacent pixels are examined and merged if their intensity values differ by less than a threshold. This threshold is increased slightly through the course of several scans of the image until the total number of distinct regions is small enough to be handled by the more optimal region merging algorithm.

Optimal region merging is implemented by computing a measure of similarity between every pair of adjacent regions and by inserting these pairs into a queue. The more nearly alike the interior properties of the two regions are, the nearer to the front of the queue they lie. Hence, the pair of regions which are most similar is always at the front of the queue. When the two regions in this pair are merged, the queue is updated by deleting all other pairs involving the two regions just merged, typically about twelve, and inserting the pairs formed using the new enlarged region. Hence, the number of pairs in the queue is reduced by three or more each time a merging takes place. The process of successive merging can be programmed to stop either when a specified number of regions remain or when the pair of regions at the front of the queue are sufficiently dissimilar.

The operations used here are thresholding, table entry and table look up.

The Brice and Fennema Region Grower

This method uses a pre-partitioned region map and attempts to merge the regions so that ultimately the region boundaries conform to the natural lines of the scene [5]. The algorithm attempts to produce a result which is void of any "false" partitions caused by image quantization and noise. The use of regions as the data type allows global constraints to be applied.

An initial partition is constructed from a normal gray scale image by grouping pixels with uniform gray levels into regions. There are usually boundaries due to optical effects, shadows, non-uniformities of object surfaces, etc. The strategy of the algorithm is to merge regions according to global criteria using the so called Phagocyte heuristic. Then a localized boundary strength criterion (Weakness heuristic) merges similar adjoining regions. The Phagocyte heuristic guides the merging by smoothing and

shortening the resulting boundary. It joins regions that are separated by a weak boundary and if it does not grow too fast. Given two regions to be merged the following operations are performed:

1. Differences in gray scale between the two regions are computed along their common boundary. Those with values less than a pre-designated threshold form the "weak" boundary.

2. The length of the weak boundary in 1) is compared with the minimum of the perimeters of the two regions. If this ratio is greater than some threshold, the regions are merged. This is related to how the boundary changes in length as the regions are merged. If it is greater than .5, the boundary must shrink; if it is less than .5, it must lengthen.

The Weakness heuristic joins regions solely on the basis of the boundary strength separating them. The percentage of weak points in the intersecting boundary is computed. If it exceeds some threshold, the regions are merged.

The basic operations used are thresholding and table lookup.

2.1.4.5 Texture Handling Algorithms

Normal region growing and edge finding algorithms cannot effectively segment a scene with high occurances of intensity changes, as would be apparent, for instance, in a picture of a field of grain. Special algorithms have been developed to deal with texture information in order to perform meaningful scene segmentation.

Autocorrelation

Almost all textures consist of an alternation between bright areas and dark areas. Hence, one useful measure of the texture of an area would be the periodicity of these alternations at various orientations. This can be found

by computing the autocorrelation function of the grayscale values in that area [19]. If g(x,y) is the grayscale value at coordinate (x,y) in the image, and if N is the neighborhood over which the autocorrelation is to be integrated, then for each (i,j) in N:

$$r(i,j) = \sum_{xy} g(x,y)*g(x+i,y+j) \; / \; \sum_{xy} g(x,y)^2 \; .$$

The values of i and j for which r(i,j) is high indicate the approximate values of periodicity present in the texture.

This process can be visualized as shifting a copy of the neighborhood i pixels horizontally and j pixels vertically and multiplying point by point with the unshifted version. Haralick added a further refinement that only points where the neighborhood and its shifted version overlapped would be included, whence it became necessary to normalize the result by dividing r(i,j) by the fraction of the area occupied by the shifted copy of the neighborhood, i.e., the overlap.

The basic operation used here is cross correlation.

Run Length Statistics

Galloway [20] used the lengths of runs of different grayscale values to classify textures. Four different orientations of runs were used, 0, 45, 90, and 135 degrees. For this, each line of the specified orientation has to be scanned, and every unbroken sequence of j pixels of intensity i is recorded by incrementing p(i,j). This histogram of various run lengths of various intensity values was condensed down into five parameters as follows. Scaling each p(i,j) by the inverse square of the run length j and by the square of j emphasized short and long runs, respectively. Variation in intensity values

and in run lengths was obtained by squaring the sum over all run lengths and intensities, respectively, before summing over the other parameter. The fifth parameter was obtained by dividing the total volume under the histogram by the sum of j*p(i,j) over the whole histogram. These five parameters were computed for each of the four orientations yielding a total of twenty components in a vector describing the texture.

The operations used here are vector inner product and histogramming.

2.1.4.6 Image Registration Algorithms

Image registration algorithms are used to locate corresponding points in a sequence of images. These images may be pairs of images from binocular cameras, or successive frames of images from a motion picture camera. The alignment of the image pairs allows the computation of depth for the surfaces of objects in the pairs, and allows the computation of motion information.

Image Registration (Nevatia)

The Nevatia method uses motion stereo to make a more reliable depth measurement of a region in an image [21]. There is less computational effort, even though more than two stereo views are analyzed. The main problem is that of correspondence, i.e. locating corresponding points in stereo views. The depth calculation (triangulation) is more accurate for larger stereo angles. However, the disparity, or the displacement of the point of interest, increases. Thus a larger area of the image has to be searched for to find the corresponding object. This problem can be alleviated by using successive intermediate stereo views, which do correspondence over a smaller area.

The procedure is as follows:

1. Given a stereo angle, determine the number of intermediate views (K).
2. Project the monocular ray of the first stereo view onto the second stereo view. Searching is done along this line in a rectangle of dimension N by M using correlation by mean square differencing.
3. In each intermediate stereo view search for the object within a N*M/(K-1) neighborhood of integer displacements, but with the last computation being done in real arithmetic.
4. Use triangulation to create a depth map.

Experiments show that using a 333 by 256 digitizer, an accuracy of 2.54 mm at an object distance of 1 m can be achieved. A typical computation time for this method is 10 seconds on a PDP-10, although it varies greatly on the number of stereo views taken.

The basic operations are correlation and triangulation.

Image Registration by Template Matching

Template matching to locate objects in an image is used in landmark registration of weather photos [22]. The earth's disk in an image can be aligned with that of a previous image:

1. by locating a pair of edges at the earth-space boundary,
2. by defining a slope of a chord connecting the edge pair, and
3. by making vertical adjustments to match the chords of both images.

This can be extended to general pictures with circular, well defined boundaries. The idea is to minimize the difference between two features, in this case, chord slope.

Template matching is applied in conventional raster scan fashion. Doing this for each pixel is too costly in computation and time, so an initial pass is performed in "coarse" mode. Rectangular windows are convolved in such a way that they overlap the previous one by one-half their horizontal/vertical dimensions. A measure of correlation between this window and the template is computed as a "distance". It hopes to find a perfect match (zero distance) and terminates after finding its goal. However, such a case rarely occurs and the best candidate (one with the smallest distance) does not always correspond to the true position. The distance values for all windows are ranked, and are inspected starting with the best candidate. A local cross operator is applied as a "fine" mode correlator. It computes correlations at its current position and four non-diagonally adjacent neighbors. Its new position is the one with the smallest distance computed. The process terminates when it does not change in position. The next highest window is analyzed in the same way.

Registration by template used in earth-satellite photos have shown to be very reliable, locating a physical landmark "perfectly" each time. It can be extended generally to images with unique objects on a uniform background. Its performance is overwhelmingly better than a brute force template match.

The basic operation used is correlation.

2.1.4.7 Image Compression Algorithms

Image compression algorithms are applied to raw data to reduce transmission bandwidths and to provide data reduction for storage in digital form.

Transform Coding

When taking the Fourier or Hadamard transform of an image, most of the energy in the transformed image will almost always be concentrated at the lower spatial frequencies. One can take advantage of this by allocating fewer bits to transmit the high spatial frequencies, since there is less information there. Results using both Fourier and Hadamard transforms indicate nearly identical performance, but the Hadamard transform is significantly easier to compute [23]. Hence, only the Hadamard transform will be considered here.

The image is usually broken up into small blocks, each of which will be separately encoded. To simplify the calculations, the block size is restricted to powers of two, e.g., eight by eight is used below. Then each row, consisting of the eight samples $g_0, g_1, g_2, \ldots, g_7$, is transformed as follows.

first pass	second pass	third pass
$h_0 = g_0 + g_1$	$g_0 = h_0 + h_2$	$h_0 = g_0 + g_4$
$h_1 = g_0 - g_1$	$g_1 = h_0 - h_2$	$h_1 = g_0 - g_4$
$h_2 = g_2 + g_3$	$g_2 = h_1 - h_3$	$h_2 = g_1 - g_5$
$h_3 = g_2 - g_3$	$g_3 = h_1 + h_3$	$h_3 = g_1 + g_5$
$h_4 = g_4 + g_5$	$g_4 = h_4 + h_6$	$h_4 = g_2 + g_6$
$h_5 = g_4 - g_5$	$g_5 = h_4 - h_6$	$h_5 = g_2 - g_6$
$h_6 = g_6 + g_7$	$g_6 = h_5 - h_7$	$h_6 = g_3 - g_7$
$h_7 = g_6 - g_7$	$g_7 = h_5 + h_7$	$h_7 = g_3 + g_7$

After each row has been transformed, exactly the same is done to each column in the block. This particular algorithm, which yields the transform coefficients in order of increasing sequence (analogous to frequency in the Fourier transform), was given in [24].

After the matrix of transform coefficients is obtained, the coefficients containing most of the energy are encoded, while the rest are either thrown away or encoded with just a few bits. The simplest approach is to pick a

standard set of coefficients to always encode which experience has shown to work well. A more sophisticated approach which selects which coefficients to encode can also be used but has the disadvantage of requiring extra bits in the encoding to specify which coefficient is being conveyed.

The basic operation here is the FFT.

Differential Pulse Code Modulation

Since the changes in intensity from one pixel to the next are usually quite small compared to the total number of gray levels, significant data compression can be achieved by encoding the differences between successive pixels. The simplest scheme only allows a fixed set of differences to be transmitted using a fixed block length [23]. For example, if four bits are allowed per pixel, then only differences ranging from -8 to +7 can be transmitted. A more sophisticated scheme uses a variable length code. Here, the differences which occur most frequently are encoded using the shortest code words, while the longer code words encode the differences which occur only occasionally. Such a code might look like the following:

difference	codeword	difference	codeword
0	00		
1	010	-1	011
2	100	-2	101
3	1100	-3	1101
4	11100	-4	11101
etc.		etc.	

Such codes must always have the property that no code word be identical to the beginning of some other code word, or else the decoder might incorrectly determine the boundaries between code words. Transmission of such a code is generally done using a lookup table to convert the differences into their

corresponding code words, so execution is very fast.

The operations used here are differencing and table lookup.

2.2 Important Signal Processing Kernels in Radar Electronic Warfare, Communication and Image Processing Systems

This section summarizes work that was done in identifying and mathematically characterizing important signal processing kernels that became apparent during the system studies described in Section 2.1. Of particular interest is the summary list given in Section 2.2.9. Although at this time the list is not exhaustive, we believe that the most important kernels are represented there.

2.2.1 Unitary Transformations

This class of transformations includes most (if not all) of the important transforms used in digital filtering and image compression. A unitary transformation is characterized by a unitary matrix whose rows (columns) are orthogonal vectors. The class is described as the multiplication of a matrix and a vector. The Karhunen-Loeve, Hadamard, discrete Fourier, Haar, slant, and number theoretic transforms are familiar members of this class.

Unfortunately, it is not sufficient simply to have a machine that can execute fast matrix multiplication, because efficient transform computation often requires an algorithm that takes advantage of the special structure of the $\hat{A}$ matrix. For example, the FFT algorithm is actually a sequence of matrix multiplications, where the matrix that characterizes each stage has a simple sparse form. Similarly, the fast Hadamard transform takes advantage of the fact that all entries in the Hadamard matrix are ± 1. In structure, a

Fermat number transform is identical to the FFT. However, the arithmetic is modular integer arithmetic rather than complex arithmetic. In this case the control of the data flow is identical to the FFT, although the hardware details in the ALU are significantly different.

A. General Formulation: $Y(k) = \sum_{n=0}^{N-1} a_{kn}\, y(n)$, $k=0,\ldots, N-1$.

Let

$$\bar{Y} = \begin{bmatrix} Y(o) \\ \cdot \\ \cdot \\ \cdot \\ Y(N-1) \end{bmatrix} \quad , \bar{y} = \begin{bmatrix} y(o) \\ \cdot \\ \cdot \\ \cdot \\ y(N-1) \end{bmatrix}$$

$$\tilde{A} = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ a_{N1} & \cdots & a_{NN} \end{bmatrix}$$

$\Rightarrow \bar{Y} = \tilde{A}\, \bar{y}$ (Multiplication of a matrix times a vector.)

B. Special Cases

1. DFT $\quad F(k) = \sum_{n=0}^{N-1} e^{j\frac{2\pi}{N} nk} f(n)$

2. NTT $\quad F(k) = \left| \sum_{n=0}^{N-1} \alpha^{nk} f(n) \right|_{\text{mod } M}$

3. WHT $\quad$ (refer to section 2.1.3)

2.2.2 Convolution/Correlation/Interpolation

Convolution and correlation are members of the same generic class because they each involve forming the inner product of a system vector and the data vector. For convolution the system vector is often a finite length impulse response of a digital filter. For correlation, the system vector is a stored reference waveform. Interpolation differs somewhat in that the system vector is time varying. However, high speed computation of inner products is the

basic operation required of this class.

A. Finite Convolution

$$y(n) = \sum_{k=0}^{N-1} h(k)\, x(n-k)$$

$$\bar{h} = \begin{bmatrix} h(0) \\ \cdot \\ \cdot \\ \cdot \\ h(N-1) \end{bmatrix} \quad , \bar{x}(n) = \begin{bmatrix} x(n) \\ \cdot \\ \cdot \\ \cdot \\ x(n-N+1) \end{bmatrix}$$

$$\boxed{y(n) = h^{-T}\, \bar{x}(n)}$$

B. Correlation (finite length)

$$r(n) = \sum_{n=0}^{N-1} x(n)y(n+k)$$

$$\bar{x} = \begin{bmatrix} x(0) \\ \cdot \\ \cdot \\ \cdot \\ x(n-N+1) \end{bmatrix} \quad , \bar{y}(n) = \begin{bmatrix} y(k) \\ \cdot \\ \cdot \\ \cdot \\ y(N+k-1) \end{bmatrix}$$

$$\boxed{r(n) = x^{-T}\, \bar{y}(n)}$$

C. Interpolation (FIR filter)

$$y(t_i) = \sum_{k=-\left[\frac{N-1}{2}\right]}^{+\left[\frac{N-1}{2}\right]} f(t_i,k)\, x(n-k) \quad i = 0,\ldots, M-1$$

This equation can be interpreted to be M time invariant filters, or as a single time varying filter.

$f(t_i,k)$ - interpolation functions (filters).

$$\bar{f}(t_i) = \begin{bmatrix} f(t_i,+\left[\frac{N-1}{2}\right]) \\ \vdots \\ f(t_i,-\left[\frac{N-1}{2}\right]) \end{bmatrix}, \quad \bar{x}(n) = \begin{bmatrix} x(n-\left[\frac{N-1}{2}\right]) \\ \vdots \\ x(n+\left[\frac{N-1}{2}\right]) \end{bmatrix}$$

$$\boxed{y(t_i,n) = f^{-T}(t_i) \cdot \bar{x}(n) \quad i=0,\ldots,M-1}$$

2.2.3 Recursive Difference Equations

This class includes both time invariant and time-varying IIR digital filters and algorithms for updating the coefficient values of adaptive filters. A difference equation can be characterized as the difference of two inner product operations. A recursive operation has fundamental differences as compared to a nonrecursive operation because quantization error, noise, or range overflow errors are fed back and compounded. Wordlength and number representation are very important in a machine that must implement recursive equations in a real time environment.

A. General Formulation

$$y(n) = \sum_{k=0}^{N-1} a_k\, u(n-k) - \sum_{k=1}^{N-1} b_k y(n-k) \quad \left\{\begin{matrix}\text{Time} \\ \text{Invariant}\end{matrix}\right\}$$

or

$$y(n) = \sum_{k=0}^{N-1} a_k(n)\, u(n-k) - \sum_{k=1}^{N-1} b_k(n) y(n-k) \quad \left\{\begin{matrix}\text{Time} \\ \text{Varying}\end{matrix}\right\}$$

$$\bar{a}(n) = \begin{bmatrix} a_o(n) \\ \vdots \\ a_{N-1}(n) \end{bmatrix}, \quad \bar{b}(n) = \begin{bmatrix} b_1(n) \\ \vdots \\ b_{N-1}(n) \end{bmatrix}$$

$$\bar{u}(n) = \begin{bmatrix} u(n) \\ \vdots \\ u(n-N+1) \end{bmatrix}, \quad \bar{y}(n-1) = \begin{bmatrix} y(n-1) \\ \vdots \\ y(n-N+1) \end{bmatrix}$$

$$y(n) = a^{-T}(n) \cdot \bar{u}(n) - b^{-T}(n)\bar{y}(n-1)$$

(Difference of two inner products)

B. Features

1. Adaptive filters are an important class of time varying filters.

2. A multiplexed filter is a periodically time varying filter.

2.2.4 Combinatorial Vector Inner Product

This algorithm, which has widespread usage in many types of signal processing, takes advantage of the fact that the system vector is often a fixed constant vector. A precomputed table of all linear combinations of the elements of the system vector is stored, so that an inner product can be

formed by successive shifting, memory fetching, and adding. This basic machine cycle must be computed n times, where n is the number of bits in the signal samples.

A. General Formulation

Assume that the system vector $\bar{a}$ is time invariant, or a "constant" parameter.

$$y(n) = \sum_{k=0}^{N-1} a_k \, x(n-k)$$

$$\text{Let } x(n-k) = \sum_{\ell=0}^{N-1} 2^{\ell} x_{\ell}(n-k) \quad \left\{\begin{matrix}\text{Binary Integer} \\ \text{Representation}\end{matrix}\right\}$$

$$x(n-k) \cong \underbrace{x_{N-1} \ldots x_o(n-k)}_{\text{binary word}}$$

Then

$$y(n) = \sum_{\ell=0}^{N-1} F(A_{\ell}(n)) 2^{\ell}$$

where $A_{\ell}(n) = x_{\ell}(n-N+1) \ldots x_{\ell}(n)$ (address)

and $F(A_{\ell}(n)) = \sum_{k=0}^{N-1} a_k \, x_{\ell}(n-k)$ (stored function)

B. Features

1. The inner product is computed without multiplication.
2. A precomputed stored function F(.) is required.
3. The amount of memory required for F(.) is 2^N words.
4. The amount of computation needed to compute F(.) is what is necessary to compute <u>all possible</u> linear combinations of the N a_k's, or all possible linear combinations of elements of the system vector.
5. The basic operator cycle is:

$$y_{\ell}'(n) = 2\, y_{\ell-1}'(n) + F(A_{\ell}(n))$$

2.2.5 Vector Norms

The calculation of a vector norm is important in many digital processing applications. The most general formulation of the norm is in terms of the p norms from complex analysis. The p = 2 norm is essentially the energy content of a signal; the $p = \infty$ is the maximum value of the sequence. Norm calculations differ from the other generic classes because they are operations on a single vector and they often require taking roots. In many situations the norm calculations are not done in real time, but rather are done on stored data during an analysis phase.

$$||x(n)||_p = \left[\sum_{k=0}^{N-1} |x(k)|^p\right]^{1/p} \quad \text{(p-norm)}$$

$$\langle||x(n)||_p\rangle = \left[\frac{1}{N}\sum_{k=0}^{N-1} |x(k)|^p\right]^{1/p} \quad \text{(average p-norm)}$$

The average p-norm, denoted $\langle||x(n)||_p\rangle$, is similar in computational form to the p-norm, differing only in the 1/N factor inside the root-p operation. For p=2, $\langle||x(n)||_p\rangle$ corresponds to the RMS value of x(n). For a zero mean process, $\langle||x(n)||_p\rangle$ corresponds to the variance. Therefore, it appears that the operations of first and second order statistics are characterized analytically by the p-norm representation.

By referring back to the discussion of EW systems in Section 2.1.2, it can be seen that the error metrics (least absolute, least squares, least infinite) can also be described as modified forms of the p-norm calculation. The error metrics normally have a weighting function w(n) that is applied prior to the calculation of the the norm, but this represents a minor modification in the basic algorithm.

2.2.6 Threshold Operations

The outcome of an operation often depends on how a computed value relates to some prescribed value, called a threshold value. Typically the threshold value is chosen based on some a priori information about the data on which the operation is performed. Although it is such a simple operation, it is a fundamental operation in image processing and in the other signal processing operations. It has the following general form:

$$g(x) = \begin{cases} h(x) & \text{if } h(x) \geq \theta \\ k & \text{otherwise} \end{cases}$$

where θ is the threshold value and $h(x)$ is the value computed by some operation on x. k is a fixed constant, typically 0.

2.2.7 Histogram Operations

The compilation of histograms is another operation which occurs frequently in signal processing. In image processing it is used to consolidate intensity information to be used for determining frequency distributions and for deciding threshold values. Histograms can be generated as data is coming into the system. For instance, in image processing the intensity level can be used to index the histogram table at the same time it is being stored in an image array. The general form of the operation is:

$$H(v) = H(v) + 1$$

where v is the intensity value and H is a vector of length n.

2.2.8 Two-Dimensional Kernels

All of the previous kernels have 2D (or multidimensional) counterparts that are often direct extensions of the 1D concepts. Work is currently proceeding to investigate important nonseparable 2D kernels that may present unusual computational difficulties. It was found previously that with respect to 1D sequence processing, many of the fundamental operations can be characterized as matrix-vector multiplication or as inner product computations. However, with respect to 2D sequence processing, this simple characterization is simply not possible. The classic problem of transposing a 2D matrix that is stored in secondary memory is an example of how an operation can be completely dominated by the complexities of addressing data blocks that are too large to fit into memory. In 2D, a separable unitary transformation becomes a sequence of 1D transformations along the rows, a matrix transposition, and a second sequence of 1D row transformations. For the computation of a 2D FFT, often the matrix transposition will require 90% or more of the total execution time. This suggests that 2D problems should be placed in one of two categories: 1) those that use data blocks that fit entirely into primary memory, and 2) those that require secondary memory (disk, tape, drum, etc.). The first class of problems will have many similarities with 1D signal processing. The second will have special problems that will be addressed during the course of future research studies.

2.2.9 Summary of Kernels for Signal Processing and Image Processing

2.2.9.1 Summary of Signal Processing Kernels

I. Unitary Transformations
   - A. Data Compression Kernels
     1. discrete Fourier (FFT)
     2. Walsh-Hadamard transform
     3. Haar Transform
     4. Slant transform
     5. sine-cosine transform
   - B. Convolution Transforms
     1. discrete Fourier transform (FFT)
     2. Number theoretic transforms
   - C. Spectral Transformations
     1. discrete Fourier transform (FFT Cooley-Tukey)
     2. chirp-z transformation
     3. Winograd FFT
     4. Cordic FFT

II. Linear Filtering
   - A. Finite Convolution
     1. FIR digital filters
     2. Sliding window integrators
     3. Matched filters
   - B. Finite Correlation
   - C. Interpolation
   - D. Adaptive Recursion
   - E. Non-Adaptive Recursion
   - F. Combinatorial Vector Inner Product
   - G. Linear predictive coding
   - H. Residue number arithmetic
   - I. Decimation filtering

III. Single Vector Operations
   - A. Windowing
   - B. Mean
   - C. Variance
   - D. Power spectrum
   - E. Autocorrelation
   - F. p-norm metrics
   - G. Maximum element search
   - H. Minimum element search

IV. 2D Operations
   - A. 2D Finite Convolutions
   - B. 2D Finite Correlations
   - C. 2D Unitary Transformations (2D FFT, 2D WHT, etc.)
   - D. 2D Linear Recursion
   - E. Coordinate transformations
   - F. Matrix Transposition
   - G. 2D Interpolation

2.2.9.2 Summary of Image Processing Kernels

I. Convolutions
   - A. Edge Finding Algorithms
     1. Compass Gradient
     2. Roberts
     3. Sobel
     4. Burr
     5. Kirsch
     6. Wallis
     7. Hueckel
   - B. Edge Linking
   - C. Transform Coding (FFT)

II. Norm Operations
   - A. Distance Computations
     1. Roberts
     2. Sobel
     3. Burr
     4. Hueckel
   - B. Mean Square Difference (Yakimovsky)
   - C. Edge Linking
     1. Ramer
     2. McKee and Aggarwal
   - D. Image Registration
   - E. Image Compression (DPCM)

III. Table Lookup and Entry
   - A. Edge Finding
   - B. Edge Linking
     1. Ramer
     2. Hough
     3. McKee and Aggarwal
   - C. Region Growing (Brice and Fennema)
   - D. Image Compression (DPCM)

IV. Threshold Operation
   - A. Edge Finding
   - B. Region Growing
   - C. Edge Linking

V. Interpolation (Burr)

VI. Histogram Generation
   - A. Edge Finding
     1. Yakimovsky
     2. Hough
   - B. Region Growing (Ohlander)
   - C. Run Length Coding

VII. Cross Correlation Operations
   - A. Texture Analysis
   - B. Image Registration
   - C. Template Matching

2.3 Commonality of Signal Processing Functions

The results of studies conducted in the analysis of generic avionics signal processing and image processing systems suggest that there are many common requirements that can be used to influence processor architecture. The following discussion first points out the commonality among the signal processing functions in radar, communications and electronic warfare. Then the commonality between the image processing functions and the signal processing functions is discussed. Finally, the commonality among all four areas is summarized in tabular form.

Probably the most easily identified common kernel among the signal processing functions is the discrete Fourier transform (DFT) as can be seen in Section 2.2.9.1. In radar systems the DFT is used in Doppler processing and in 2D stretch processing for synthetic aperture radars. In communications the DFT is used for data compression, noise suppression filtering and detection filtering. In electronic warfare systems the DFT is used for pulse (object) identification and for digital filtering. A common digital processor would be required to efficiently calculate DFT's by means of one of the many FFT algorithms. The processor should be on-line programmable for different block lengths. It should execute high speed complex arithmetic, and probably would incorporate specialized addressing schemes to accomodate the "decimation" indexing of the popular FFT algorithms. In high speed applications (radar and EW) the processor could operate more efficiently with stored sine and cosine tables, so that the exponential reference functions could be read from read-only memories, rather than being computed on-line.

A second important kernel required in all avionics signal processing systems is finite convolution (correlation). Presuming filters, roughing filters and noise suppression filters in synthetic aperture radars are often

linear phase FIR filters. For long finite length convolutions, the FFT algorithm can be applied by transforming the data, block at a time, multiplying the transforms point by point, and then inverse transforming the resulting spectrum. When this type of FFT-implemented convolution is applied to real time processing, considerable overhead cost is involved in buffering the data blocks and in applying the overlap-save or overlap-add algorithms required to use the circular convolution of the FFT for implementing the desired linear convolution. For short convolutions, a more direct computation may be useful since it eliminates the overhead associated with block processing. For direct convolution a high speed real-time multiplier is required (at minimum). Since radar and communication signal processing is often done in the baseband with complex (I and Q) data, high speed complex multiply also appears to be important.

Generic kernels associated with decimation and interpolation are important in all of the systems studied. Decimation and interpolation are used abundantly in communications systems as bandwidth varies, in order to maintain the minimum data rates necessary to characterize the waveforms. Two-dimensional interpolation is very important in generating high resolution imagery with spotlight mode synthetic aperture radar. Since these kernels can be expressed as linear filtering and sample rate alteration operations, they are very similar in nature to convolutional FIR digital filters. However, as pointed out in Section 2.2.2, interpolation filters are really time varying systems. Also, they may be realized by recursive filter structures. For these reasons, decimation and interpolation should be considered as distinct kernels that have widespread usage in avionics systems.

The combinatorial vector inner product (CVIP) operation represents one of the most important kernels that appear to have widespread utility. It is a very efficient technique for computing vector inner products when one of the

vectors is a fixed (time-invariant) system parameter. The CVIP has applications in radar, communications and EW systems as a generic technique for computing weighted summations of signal samples. It has even been applied (in the literature) for the implementation of a Cooley-Tukey pipelined FFT unit.

Another frequently encountered kernel is the p-norm operator, as well as the various modifications of the p-norm operator that are discussed in more detail in Section 2.2.5. The p-norm kernel is an essential ingredient in the computation of first and second order statistics, in the calculation of metrics in EW signal classification and in the calculation of metrics for target identification and map matching in radar systems. The $p=1$, $p=2$, and $p=\infty$ norms are the most commonly encountered forms. Each requires a distinct sequence of arithmetic operations. Therefore, it appears the computational efficiency of these different norms is dependent on more specific hardware structure of the processor.

Many other signal processing kernels have been identified as needed in the various systems that were studied. These include sliding window integration (a simple form of finite convolution), threshold detection, table look-up functions, matrix-vector multiply, matrix inverse, and 2D matrix transposition with secondary storage. It is important to note that the objective of this present work is to identify common basic (generic) signal processing functions, rather than to analyze specific algorithms for computing these kernels. For example, the DFT is undoubtedly a generic kernel; probably the most important one. But the DFT can be computed by the Cooley-Tukey FFT, the CORDIC FFT, the Winograd FFT, or the chirp-z transform. Identifying generic kernels is clearly only a first step. Further efforts will be needed to analyze alternative algorithms for implementing the kernels to determine the most advantageous forms or avionics system requirements.

The summary of image processing kernels, shown in Section 2.2.9.2, indicates several areas of commonality with the signal processing kernels that were identified in the study of radar, communication and electronic warfare systems. It is apparent that convolution, norm operations, interpolation and cross correlation operations are basic to each of the four areas. In addition, the threshold operation, histogram and table lookup operations, all of which are used extensively in image processing systems, also have application in the other signal processing areas. Table 1 summarizes the kernels which are common to the areas studied.

I. Convolution
II. Correlation
III. Norm Operation
IV. Interpolation
V. Unitary Transformations (FFT)
VI. Table Look-up and Entry
VII. Thresholding

Table 1. Condensed summary of common kernels from all areas studied.

Computationally, thresholding and histogramming are quite simple. Thresholding simply compares a value with a reference value and stores a third value which is either the first value or some prescribed fourth value, usually zero. Histogramming simply increments an accumulator which is indexed by the input value being counted. Table lookup operations are very applications oriented and may include such schemes as hash addressing, linear searching, or linked lists. The basic operation is to compute the address for a data

transfer.

The results of this study are extremely encouraging because so many instances of commonality were found among radar, communications, EW and image processing functions. We feel that architectures with suitable realizations of these common kernels would greatly enhance signal processing in the avionics environment and recommend that further study should be devoted to identifying such architectures.

It should be pointed out that the functions studied in this effort are mainly those functions which operate on raw data. Second and higher order functions, which operate on the results of the operations considered, more than likely also have commonalities in the four fields. Signature analysis in EW, target identification in SAR, and object identification in image processing all have similar objectives, and hence, we feel, commonality of function. Further studies should be conducted to investigate the data structures and recognition schemes in these areas in order to isolate commonalities at that level of processing. In addition, the impact of these commonalities on signal processing architectures should be studied.

2.4 Computer Architectures for Signal Processing

As the previous sections have demonstrated, signal processing functions, though diverse in their objectives, have substantial commonality in their computation. Consequently, a well-designed architecture can perform these functions in a cost-effective manner. Ideally, all computations would be performed on a single, sufficiently flexible processor and we feel that this goal should be pursued to the extent possible. Nevertheless, our investigations to date lead us to believe that even within the same application area, the requirements of high data rate, diverse functionality

and flexibility may be mutually contradictory in a cost-effective design. Our conclusion, therefore, is that a common architecture is advantageous and probably achievable but that the design of such an architecture requires a significant research effort.

Due to the complexity of the problem, this preliminary research effort has not attempted to produce a final design for the signal processing architecture. However, we have identified certain important requirements for, and promising architectural features of this signal processing architecture. From the point of view of data transfer rates and computation rates, avionics processing has the following general characteristics: raw data is obtained from input transducers (e.g., radar antennae) at extremely high data rates, and in a series of processing steps, is transformed and compressed into a meaningful and useable form. This data is operated on by high level signal processing algorithms to achieve the signal processing objectives. Finally, this information must be converted to a form easily assimilated by a human operator (e.g., video graphics) or into the real-time I/O signals which control the input transducers (e.g., antenna rotation for target tracking). Certain information may be stored away in a database for subsequent use in operations such as target identification. This last step will often involve the expansion of data into a format suitable for the output devices.

Thus, a signal processing architecture must include both the capability for high data rate, primitive but structured computations as well as complex decision-making procedures which are invoked with a relatively lower frequency. At all levels of signal processing there is substantial diversity in the functions being performed. For instance, the low end includes operations such as inner products and FFT while the high end is exemplified by graphics operations such as scene rotation, hidden line generation, shading, etc., on the one hand and searching, sorting, and information retrieval in

databases on the other hand. Although the objective should be to integrate as many of these functions as possible into one processor, it is likely that they will have to be partitioned into classes that are compatible in terms of functionality and data rate. This would permit the design of cost-effective processing elements specialized to the particular needs of each of these classes. It should be emphasized that this partitioning does not interfere with the exploitation of commonality; any given processing element will perform those functions best suited to it from across all signal processing applications.

A successful signal processing architecture will result from a design strategy that matches the structure and performance of the architecture to the nature and requirements of the signal processing task. In general, this architecture will consist of a heterogeneous collection of processing elements, memories of various speeds and capacities and buses of differing data rates. The selection of processing elements and memories with respect to their functionality and performance as well as the topology of their interconnectivity must reflect the structure of the signal processing task. Higher speed memory must be allocated to the more frequently used data sets; often executed functions must be supported by high performance processing units; high data rate computations must be facilitated by (possibly) dedicated buses between the corresponding memories and functional units.

The use of such a design strategy requires that a careful analysis be made of the signal processing task. For these purposes, the task must be represented as a network of nodes (subtasks) with data flowing along the arcs between the nodes. The nodes should be characterized by the amount and nature of computation that they represent and any other statistics that are relevant in the design strategy described above.

One approach would be to cluster nodes with common functionality and to provide, for that cluster, a processing unit with a computation capacity adequate to the needs of that cluster. On the other hand, such a clustering might lead to excessive data transfer requirements. Ideally, the clustering should be such that high bandwidth arcs lie within a cluster and only arcs of relatively low data rates cross the cluster boundaries, thereby minimizing the number of high bandwidth buses needed. Due to the asynchronous parallelism within the system, buffering is needed on the arcs between nodes. The clustering specifies the interconnectivity between the buffers (memories) and the processors, viz., processors and memories in the same cluster must be tightly coupled with short access times while the inter-cluster arcs would correspond to low bandwidth buses with low priority memory ports.

In the context of cost-effective design, a trade-off exists between functional flexibility and performance. The highest performance processing elements cannot tolerate the overhead of instruction fetching, thus requiring that control be hardwired (or microprogrammed). This, along with the high performance requirements implies that the structure of the hardware be closely patterned upon the function being performed. This allows the use of parallelism to achieve the high performance requirements for specific functions but precludes flexibility in functionality. However, limited flexibility may be achieved by incorporating residual control which allows for a certain degree of modification of both the control as well as the configuration of the hardware.

Signal processing also includes a number of subtasks (typically at a high level) which are less regularly structured. Furthermore, effective procedures for such subtasks are evolving relatively rapidly. It would, therefore, be unwise to attempt to design specialized processing elements for the currently used procedures for these subtasks. A general purpose processing unit is

essential as one component of this signal processing architecture. This processing unit can prolong the lifetime of the architecture by being flexible enough to adapt to new developments.

2.5 Suggestions for Future Research in Avionics Signal Processing

It would be productive to continue research in several distinct categories of avionics signal processing. These areas are described below.

2.5.1 Study of Current Avionics Digital Signal Processing

The purpose of further work in this area is to establish a clear understanding of avionics signal processing requirements and to document how these needs are currently being met. The project should include an investigation of both "digital signal processors" and "data processing computers". Bandwidth and wordlength requirements should be analyzed, and special issues in the selection of current and future IC technologies should be addressed. During the study, particular attention would be devoted to methods used for achieving fault tolerance and for providing automated testing capability (if any). This project would result in a definitive evaluation of the state-of-the-art.

2.5.2 Identify and Characterize Avionics Signal Processing Kernels

Sections 2.1, 2.2 and 2.3 presented preliminary results in identifying and characterizing low-level generic signal processing functions. This research project would extend results in this area to a point where kernels could be related to machine architecture. For example, it is important to evaluate alternate FFT algorithms such as the conventional Cooley-Tukey

algorithm, the CORDIC algorithm, and the Winograd algorithm. Although each produces the same spectrum, intermediate details are considerably different.

Further work in the area would serve to evaluate alternative computational algorithms for the kernels and to identify commonality among radar, EW, communication, and image processing. Particular emphasis should be placed on EW systems, since there now appears to be a sizable body of unclassified EW literature to support the work. A synthetic aperture radar is a good model for a generic system that includes most functions of interest, i.e., data compression, data transmission, real time processing (weighting, interpolation, filtering, spectral analysis), and post processing. Post processing, which involves target detection, image enhancement, edge detection, and image compression, is closely related to image processing. In general, the SAR system if representative of signal processing functions that may be required in other systems within the avionics environment.

2.5.3 Investigate Adaptive Processing Techniques

From one point of view, it can be argued that a useful processor architecture would be one that can adapt to incoming data to "configure the machine" for an optimal solution of the assigned problem. This is the concept behind a machine structured with banks of configurable polynomial arrays (CPA's) as proposed by Adaptronics [25]. However, an adaptable machine with extensive capabilities may not perform as well as a simpler machine on simple problems. This is true because the adaption process is never ideal, and the residual full-machine capability may cause poor performance when the full-machine capability is not required. An example of this phenomena was observed [26] when a nonlinear CPA filter was compared to a linear LMS adaptive filter for cancelling additive noise. The nonlinear terms in the CPA

filter were not set exactly to zero by the adaption algorithm, and they introduced noise into the gradient approximation.

Future work in adaptive signal processing techniques is needed to evaluate the properties of a CPA architecture and to illustrate its performance on several problems (to be defined). Hardware structures for realizing a CPA architecture should be investigated, and techniques for achieving reliability in CPA architectures should be proposed and evaluated.

2.5.4 Investigate the Utility of Number Theory Techniques in Avionics Signal Processing

During the last few years there have been significant results reported in the literature indicating that certain concepts from number theory can be applied to simplify hardware and achieve high data rates in digital filters [27,28]. These include residue number arithmetic, number theoretic transforms, and the Winograd FFT algorithm. The combinatorial vector inner product algorithm can be included in this general category.

Future efforts should be devoted to investigating the applicability of these new concepts to avionics signal processing problems. The techniques are very promising because they reduce hardware complexity and improve speed. However, the number theory concepts seem to produce efficiency when applied to dedicated tasks, and as such, seem to be more useful in the signal processors, rather than in the data processing computer. The results of this work would establish whether the ideal avionics signal processor is too general for number theoretic techniques, or whether the techniques can be used to advantage.

2.5.5 Investigate Avionics Visual Image Processing

Current serial computers are poorly matched to image processing problems and algorithms. At the early stages of processing especially, most operations (e.g. gradient finding) are local and independent, ideal candidates for parallel implementations, but serial computers cannot take advantage of this inherent parallelism. Architectures designed with image processing in mind could provide dramatic improvements in the performance (one or more orders of magnitude) and could make real-time high resolution image processing feasible. At present we often must choose to either look only at small windows of an image in detail in order to obtain real-time performance, or else we must accept very long processing times for each image.

The approach we suggest to this problem is to find and verify good matches between problems, algorithms and architectures. We already know a good deal about matches between problems and algorithms for image processing, but we know much less about algorithm/architecture matches. Furthermore, much more effort has gone into investigating algorithms well suited to several computers than has gone into parallel algorithms, however appropriate they are to specific image processing problems. We believe that the best overall system performance can only come from the consideration of problems, algorithms and architectures together.

Research in this area should proceed in several stages. The first two stages should be performed concurrently and should be followed by the remaining stages which should be performed in succession. Throughout this effort, close communication should be maintained with the architecture group and the group studying radar, communication and electronic warfare.

2.5.5.1 Task Definition for the Avionics Domain

We feel that it is important to have overall goals for the eventual systems we are aiming toward. To this end, a set of scenarios should be constructed which indicate how image processing systems will be utilized in the avionics environment. From these scenarios the overall image processing tasks should be defined as specifically as possible. This includes the projected input data rates, the types of information to be obtained from the data and the time constraints in which this information is to be obtained. Several of the tasks which should be considered include the matching of a map with an aerial image, the detection of target objects in an image, and the tracking and identification of objects in an image.

2.5.5.2 Identification of the Computational Tasks

For the scenarios proposed in the previous task, the computational tasks should be determined. These tasks should be analyzed and the functions necessary to carry out the tasks identified. Data structures for the task domains should be analyzed and the computational times necessary to keep up with the data rates determined.

2.5.5.3 Analysis of Image Processing Algorithms

A more in depth study should be made of all the algorithms used in visual image processing. These algorithms should be catalogued according to the functions they perform. For instance, the Fast Fourier Transform has many algorithmic realizations. Both basic (low-level) image processing algorithms and algorithms for image analysis and decision-making (high-level should be analyzed since the low-level algorithms must provide suitable outputs for

convenient higher-level analysis. The analysis performed on each of the algorithms will determine their efficiency with respect to memory requirements, pixel accesses and computational complexity.

The low-level image processing algorithms to be analyzed include 1) the detection of shapes such as lines, corners, curvatures, surface normals, symmetry axes, closed contours, etc. and 2) the analysis and representation of features such as color, texture, texture gradients, parallax and depth, motion direction and velocity, spatial frequency, etc.

The high-level algorithms to be examined include ones to analyze 1) object groups and relations between objects, 2) lighting type, shading, shadows and highlights, 3) object models and the recognition and description of objects, 4) precise location or tracking of objects, etc.

2.5.5.4 Identification of Kernels

A subset of algorithms should be selected and analyzed in detail. Algorithms catalogued in the preceding step should be analyzed for performance on the selected tasks, and a subset of the operations which are found to be the most important and representative selected. This subset of algorithms should be further analyzed to find ways in which the algorithms can be broken up into separate processes, and to identify kernels which can be shared by several algorithms, and generalizations of kernels which may merit special consideration within the architecture. Results available at that time from the architecture study will help determine appropriate ways to break up algorithms, and may suggest some additional algorithms which should be added to the subset to be considered in detail.

2.5.6 Investigate Computer Architectures for Signal Processing

The application-oriented design effort for communications, radar and image processing should be based on a characterization of the workload which will be supplied by the research groups active in those areas. This characterization will be in the form of kernels, i.e., well defined computations that are representative of their application areas. A kernel qualifies as such if it tends to be one of the most frequent, time-consuming or expensive computations involved in that application. Taken together they should be typical of, and identify the worst-case of, a class of computations which, together, constitute most of the workload. The significance of these kernels lies in the fact that an architecture which is geared towards these kernels and executes them efficiently, is guaranteed to perform well on the workload as a whole.

Kernels that are common across multiple application areas should be given special emphasis since a maximum payoff for a given outlay can be realized by optimizing with respect to them. However, evaluation in a broader context is also necessary in order to guarantee that resulting architectures will not be so specialized to the common kernels that they become ineffective for the total workload. Therefore, while emphasizing the commonalities, consideration should be given to the dissimilarities between the various applications to ensure that the resulting design is effective and complete with respect to each application area. Ensuring sufficient generality provides the further advantage that the architecture is likely to remain effective in the face of unanticipated developments in signal processing techniques.

The kernels should be analyzed to determine certain important characteristics. The size of the data structure operated upon by a kernel has a direct impact upon the size of main memory and the amount of paging activity

required. The computational and data rate estimates will be indicative of the processing and memory bandwidths that will be needed. The desired precision and its range should also be investigated. Finally, an effort should be made to ascertain whether kernel algorithms can be restructured to perform better on novel candidate architectures, which were not adequately considered when the algorithms were initially developed.

The kernels should be analyzed to ascertain the nature and amount of parallelism present. The nature of the parallelism can be measured by the extent to which the kernels are matched to each of a set of "distilled" architectures, i.e., fictitious machines, each of which embodies the essential features of a class of architectures. The distilled architectures to be studied include single stream instruction pipeline machines, vector processors, array processors and multi-processors. In order to focus on the nature of the parallelism inherent in the kernels, it can initially be assumed that distilled architectures have unlimited parallelism. Thus, a distilled array processor can process an array of arbitrary size in one pass. However, all operations performed concurrently must be identical. Furthermore, each distilled architecture is an extreme architecture particularly suited to exploiting specific types of parallelism in the kernels. For each type of parallelism, the extent of parallelism can be measured by observing the number of resources that could be kept busy in the corresponding distilled architecture.

A realistic architecture must consist of a judicious mixture of these distilled architectures with limited parallelism. The performance of the kernels on each of the distilled architectures will indicate the types and extent of parallelism that should be provided by the recommended signal processing architecture. This information can be used to derive from the class of distilled architectures a characterization of candidate architectures

for further study.

A means of describing the benchmark kernels should be developed. The kernel descriptions will permit rapid identification of the types of parallelism present and the amounts of each type. Transformations between alternative types of parallelism will also be readily apparent. To this end, distinctions should be made between implicit and explicit parallelism, low level and high level parallelism, alternative structured data types, sustained parallelism vs. burst parallelism, and functional, memory and bus requirements which must be explicitly provided or emulated by a candidate architecture.

Various organizations and management strategies should be studied for the four types of components that make up a parallel computer: functional units, memory, control units and buses. Each organization has its particular strengths and weaknesses and tends to be attractive under certain conditions which must be well understood in order to achieve a cost-effective computing structure. In certain cases, organizations with very high performance potential may turn out to be self-defeating in practice. For instance, a lookahead processor which attempts to achieve dramatic levels of concurrency by very complex lookahead schemes may not be cost-effective.

Often, a particular objective may be realized in a number of ways each of which is appropriate in a particular context. For example, if mutually exclusive access to a shared bus is desired, two strategies suggest themselves. An asynchronous, handshaking protocol is powerful in the sense that the bus may be acquired and released at arbitrary instants in time. This may be contrasted with the simpler time-division-multiplexed bus which may only be used during pre-defined time-slots by each potential user. When bus transactions tend to be relatively long and variable in length, the former

strategy is indicated. However, when the transactions become shorter and less variable in length, the handshaking overhead becomes disproportionately large and generality must be sacrificed in favor of the latter scheme.

Such trade-offs between cost, generality, flexibility and speed exist at all levels of detail and in every design decision. A cost-effective parallel processor design must take these into account and match the capabilities and cost of each subsystem to the requirements of the computation. As a consequence, the parallel processor might consist of a variety of processing elements possessing different organizations and functional capabilities, some of which are designed to operate upon data, others to transfer data between these processing elements and yet others to perform control, synchronization and scheduling functions. Memory of varying speeds and capacities will, in general, be distributed throughout the system to provide data and program storage capability as well as to serve as a medium of communication between processing elements. Perhaps most important, is the choice of interconnection mechanisms which will determine the access proximity between each pair of processing elements and memory units. This general design will take on a particular form based upon the signal processing workload statistics. In one case, an array structure might be indicated, while in another instance a system of autonomous processors with shared memory, might be more desirable. To make intelligent choices between the various alternatives requires that the conditions be identified for which each alternative is most effective.

For the later stages of detailed architecture evaluation, extensive simulation should be carried out using the kernel descriptions developed and an evolving set of candidate architectures tracked through interactive improvement and successively more detailed evaluation. Of the simulation languages available, SIMULA [29] is perhaps the most elegant and powerful. However, SIMULA has relatively poor bit manipulation capabilities The hardware

description languages possess adequate bit manipulation capabilities [30] and are basically parallel, but in a restrictive way and with awkward syntax.

The simulation studies will require some enhancement of these capabilities to provide for rapid tracking of changes in the simulated system by guaranteeing that modifications to one portion of a system be reflected in a change to the simulation in one, and only one place. Characterization of kernel parallelism, their primitive data types and operations, will be straightforward. A variety of levels of simulation detail should be fully supported for efficient execution. These enhancements can be developed by augmenting a standard simulation facility.

This research effort would perform a thorough analysis of the signal processing kernels with an appropriate range of distilled architectures and a reduced range of explicit candidate architectures. Analytic techniques should be developed and used as far as possible, following which extensive simulation will be required. The evaluation results obtained will yield a highly cost-effective recommended signal processing computer architecture.

SECTION III

COMPUTER ARCHITECTURES FOR FINITE ELEMENT METHODS

Finite element methods are used for computer evaluation of the behavior of complex structures. They often reduce or eliminate the need for complex prototype evaluation and permit a much broader range of potential designs to be evaluated. The potential gains from using finite element methods, however, are compromised by the inefficiency of present state of the art computers when applied to finite element method computation. Individual runs are long on medium sized computers and costly on large scale computers, leading not only to a high cost for using the method, but often limiting the number of runs for an evaluation to few runs on subproblems.

Nevertheless, finite element method computation is highly structured and many of its tasks can be performed in parallel, which gives rise to the expectation that efficient new computer systems can be developed which are specialized for performing finite element method computation in a highly efficient manner. The primary objective of this research is to develop such computer structures. Intermediate objectives include the development and evaluation of algorithms for use in the finite element method, assessing their utility for alternative computer structures, developing and assessing specialized structures, evaluating total systems, and making final recommendations concerning the most effective architectures and the appropriate algorithms for them to exploit their capabilities to the fullest. Final recommendations will be accompanied by an evaluation of performance and cost-effectiveness.

In addition to the principle objective described above, this research will have broad impact on the more general problem of discovering appropriate methodologies for developing and evaluating problem-dependent architectures.

Of particular relevance for achieving this objective, are the problems of evolving appropriate techniques for tailoring architectures to problems, developing algorithms for architectures, and establishing means by which the effective algorithm-architecture pairs can be compared with one another despite their widely different approaches to solving the problem.

Thus, the emphasis on finite element method computation in this research can be viewed both as an important objective in itself as well as providing a necessary, complete and important example for the development of appropriate tools for meeting the more general objective.

A pilot study on this subject has been in progress since February 1979 under the sponsorship of the Wright-Patterson, Flight Dynamics Laboratory. The performance of existing finite element code has been measured and evaluated. The critical computations in this code perform multiplication of large matrices and solving large sets of linear equations. These kernels of code involve significant computation time which becomes dominant for large nonlinear problems. They also cause substantial paging traffic. Known algorithms for performing these kernels and existing architectural studies have been collected. Both analytical and experimental studies of the speedups achievable through parallel processing of these kernels have been initiated. Preliminary results indicate that some parallel architectures may be very attractive.

Plans for the continuation of this research include performance measurement of dynamic (time-varying) problems; further analysis of existing architectures and algorithms; direct consideration of sparse matrix, symmetric matrix, and blocked matrix approaches; and evaluation of memory organization and management strategies. Algorithms will be selected and adapted for particular architectures in light of their properties with respect

to the above considerations. Alternative strategies for running these algorithms will be evaluated. Essential primitive operations will be defined for performing such matrix-oriented computations efficiently. These will lead to recommended instruction set architectures and dedicated specialized function units. Finally, recommended architectures will be selected in the light of these analytical and experimental results. The scope of consideration will include existing large computers, moderately or extensively modified small to medium sized computers with enhanced memory and function capability, as well as special-purpose highly parallel computer or multicomputer systems.

3.1 Computer Needs in Finite Element Analysis

The finite element method can be characterized as a tool requiring large volumes of topologic and geometric data, well defined operations on large numbers of small and large matrices, many of which are sparse, large volumes of data being manipulated during the solution process, and significant problems associated with reducing and displaying the results in a meaningful way.

These characteristics translate into a need for

1. geometric modeling systems for generating problems, combined with techniques for mapping problem descriptions onto finite element meshes;
2. significant data handling capabilities capable of operating efficiently on sparse matrices;
3. facilities for handling large volumes of data efficiently;
4. extremely fast scalar and parallel processing power;
5. intelligent graphics systems for displaying both the problem and the results in a meaningful manner.

Today almost all of the sophistication needed to do large scale finite element analysis is handled at the software level. Basically, engineers are using the computer as a very primitive fast calculating device. This is due primarily to the fact that computer manufacturers have not provided engineers with anything beyond a very basic computing environment. There are no computers today which automatically handle sparse matrices. Many of the largest computers do not even have virtual memory. Graphics utilization by engineers often reduce to drawing simple pictures on a storage tube or refresh type devices. Consequently, all of the work must be done at the software level.

In some instances, in order to circumvent the lack of hardware, engineers have developed software virtual computing systems in which they then program the finite element systems. In effect, the use of these virtual software systems implies a lack of sophisticated operating systems and hardware from the manufacturer. These software systems come in many sizes and shapes. Examples are DMAP, ICES, DRS, NORSAM, POLO, etc. Basically these systems have been used to provide various levels of dynamic memory allocation and data mappings such as the sparse matrices mentioned earlier. Through these systems, various levels of sophisticated finite element software can and have been written.

Virtual systems represent a tool for software development. They make it somewhat easier to develop large complex systems. However, these tools cannot increase the throughput of a finite element program. If anything, they must slow it down. In general the tradeoff has been speed for flexibility. It would appear then that a significant increase in productivity would occur if a) a fast floating point capability were available, and b) some of the aspects of virtual software systems, such as sparse matrix operations, were available at the hardware level.

In order to determine the exact needs in the hardware, an initial study has been undertaken to determine the level of computing resources utilized in the various aspects of the finite element problem. The results of this study are presented in the following paragraphs.

3.2 Evaluation of the Finite Element Problem

The purpose of this phase of the research project is to determine how computer resources are utilized for solution of finite element problems. This approach has some serious drawbacks. In particular, in order to derive the statistics it is necessary to depend on a particular finite element code. These statistics may vary somewhat from code to code. At the same time, since the basic algorithms being used today do not vary significantly, it would seem that the trends would not change significantly between codes.

The computer code used is called FINITE, which is a general purpose structural mechanics system based on a virtual software system called POLO. Each of these software systems has been reported in the literature. FINITE is based on the availability of POLO's data base management and virtual data base facility. It operates on sparse matrices in a relatively intelligent manner; it is quite flexible relative to structural modeling. It is the level of general purpose code that one would expect to find operating on computers in the future. Thus, the statistics derived from it should be of value in developing the concepts of a new machine. POLO, which drives the FINITE system, is an interpreter, thus it presents an ideal working environment for this project. Through POLO the authors were able to determine the levels of resources used for data base management, paging, and solution of the problem.

The solution procedure was broken down into processes such as assembly, triangulation, residual load computation, etc.; individual sub-routine resource utilization was also determined. In this way we were able to determine, for example, not only how much CPU time and paging was associated with assembly of the stiffness matrix of a structure, but also how much of the assembly time was associated with the generation of individual stiffness matrices and the re-orientation of the matrices in the global space, as well as the operations associated with various topological considerations that are performed during the stiffness assembly.

For the purposes of the discussion herein, the finite element method was characterized as a nonlinear problem. Linear problems are a subset of the nonlinear case. The method was then divided into the following ten subprocesses.

1. Data generation and creation of data bases.
2. Processing of material models.
3. Generating element stiffness matrices.
4. Miscellaneous items associated with stiffness assembly.
5. Assembly of the structure stiffness matrix.
6. Decomposition of the stiffness matrix.
7. Calculation of equivalent nodal loads from loads applied as internal pressures, etc.
8. Displacement recovery for given set of loads.
9. Stress recovery for a set of displacements.
10. Calculation of residual loads for nonlinear iterations.

In the above, steps 3, 4, and 5 are associated with the calculation of the stiffness matrix. Step 4 is always assigned to the processing which is not directly assigned to either structural assembly or element stiffness generation.

3.3 Example Problems

Two example problems were solved using both linear and nonlinear analysis. The first problem is a nonlinear pressure vessel as shown in Figures 6 and 7. This problem is relatively simple and is small. It is axisymmetric and has 288 nodes, 45 elements (8 node quads), and a very narrow band. The elements are isoparametric, and thus numerical integrations must be done in order to generate the stiffness matrix. This is typical of most problems being solved today. The problem was solved for both a linear and a nonlinear response. In addition, a second nonlinear solution was obtained using substructuring. In the latter case those portions of the structure which were known to remain elastic were condensed out of the problem, and thus the iterations associated with the nonlinear process needed to operate on only a small portion of the elements in the problem. The purpose of using the substructuring was to determine if more advanced solution techniques of analysis would drastically change the statistics. If the latter were the case, it would of course be improper to base any machine architecture on statistics which were so sensitive. The results of the analyses are shown in Figures 8, 9 and 10. The lower histogram in each figure shows, for a particular problem, the percentage of CPU time spent in each of the individual processes of the analysis. Each bar of the lower histogram consists of a crosshatch region and a clear region. The crosshatch region represents CPU time used in doing "useful work," i.e., computations associated with the finite element method. The clear portions of each bar represents CPU time spent in data base management, i.e., the time required to do the mapping of the data onto the virtual space and for dynamic memory allocation. The upper histogram of each figure shows the level of paging for each process. Both the CPU utilization and the paging are expressed as percentages and therefore relatively machine independent. At the top of the figure, the pages turned

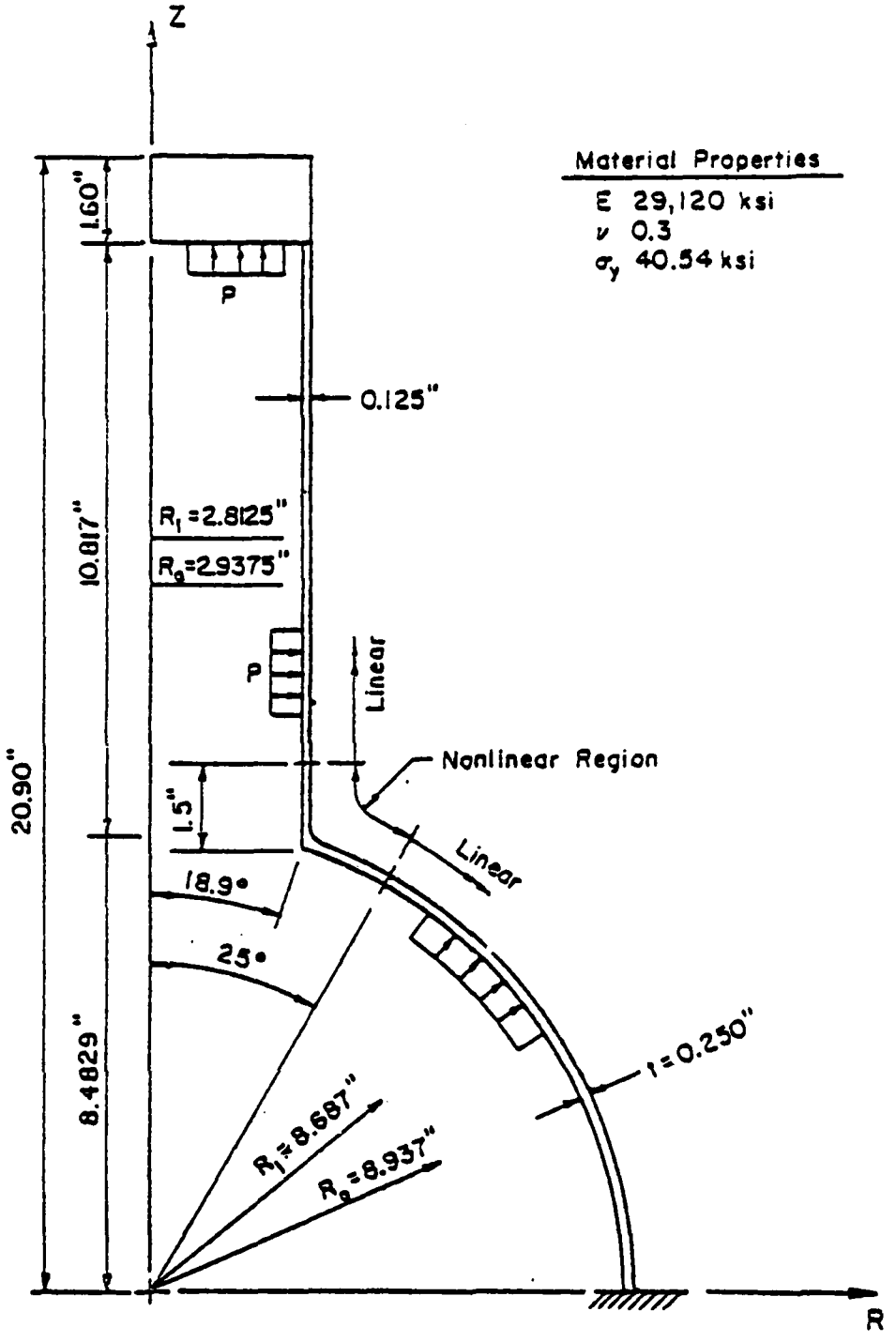


Figure 6. Axisymmetric pressure vessel.

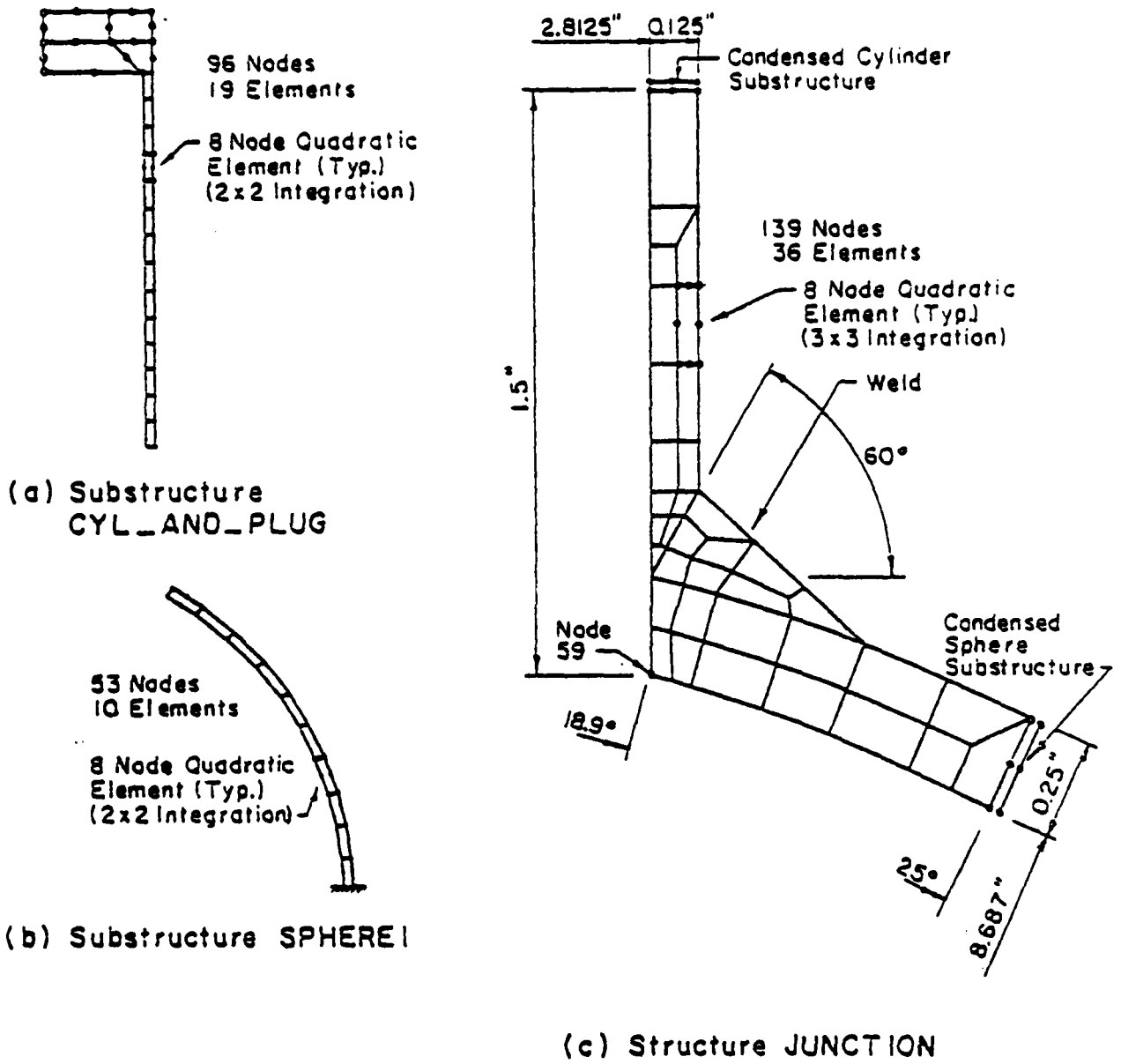


Figure 7. Axisymmetric pressure vessel finite element model.

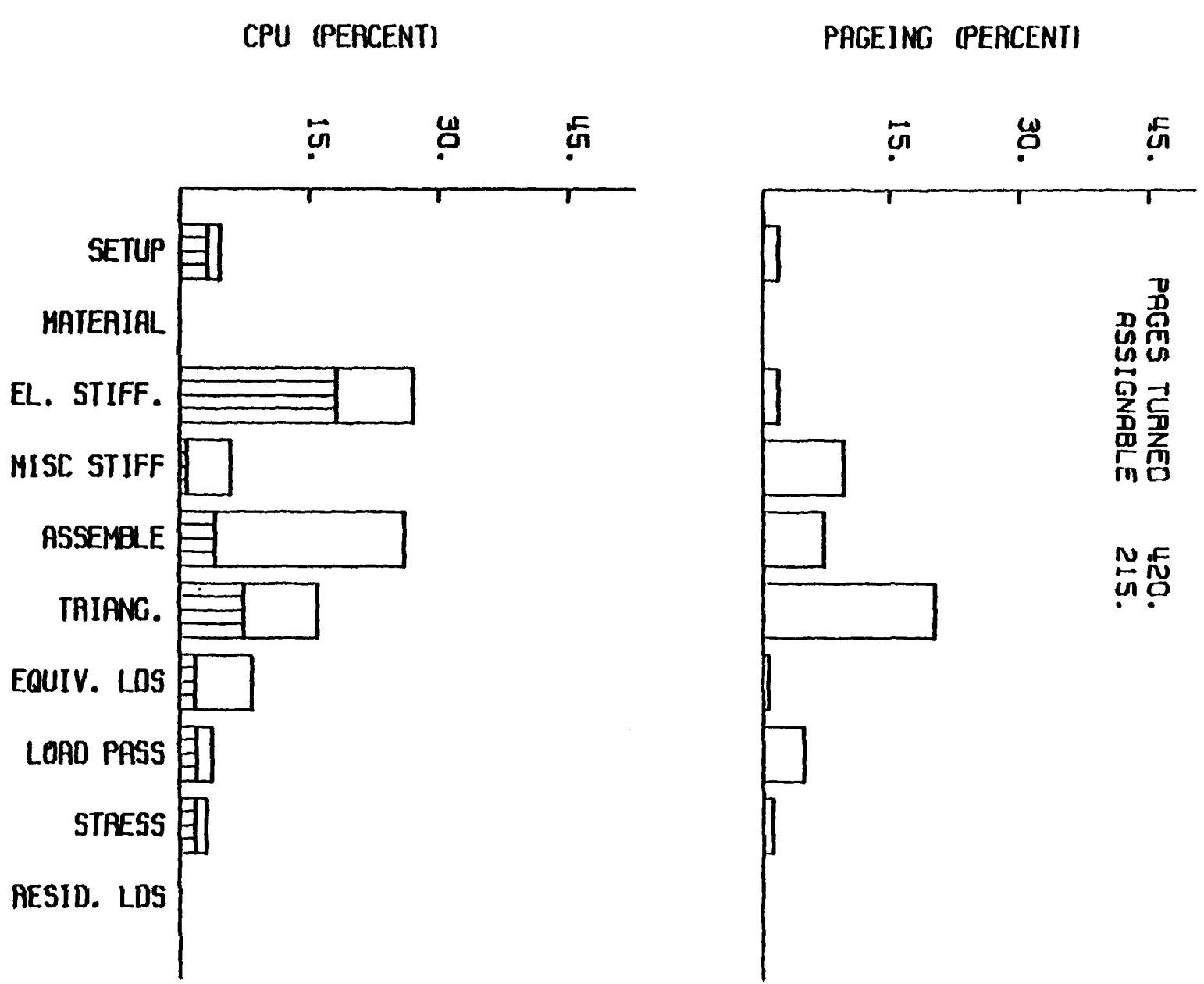


Figure 8. Linear pressure vessel.

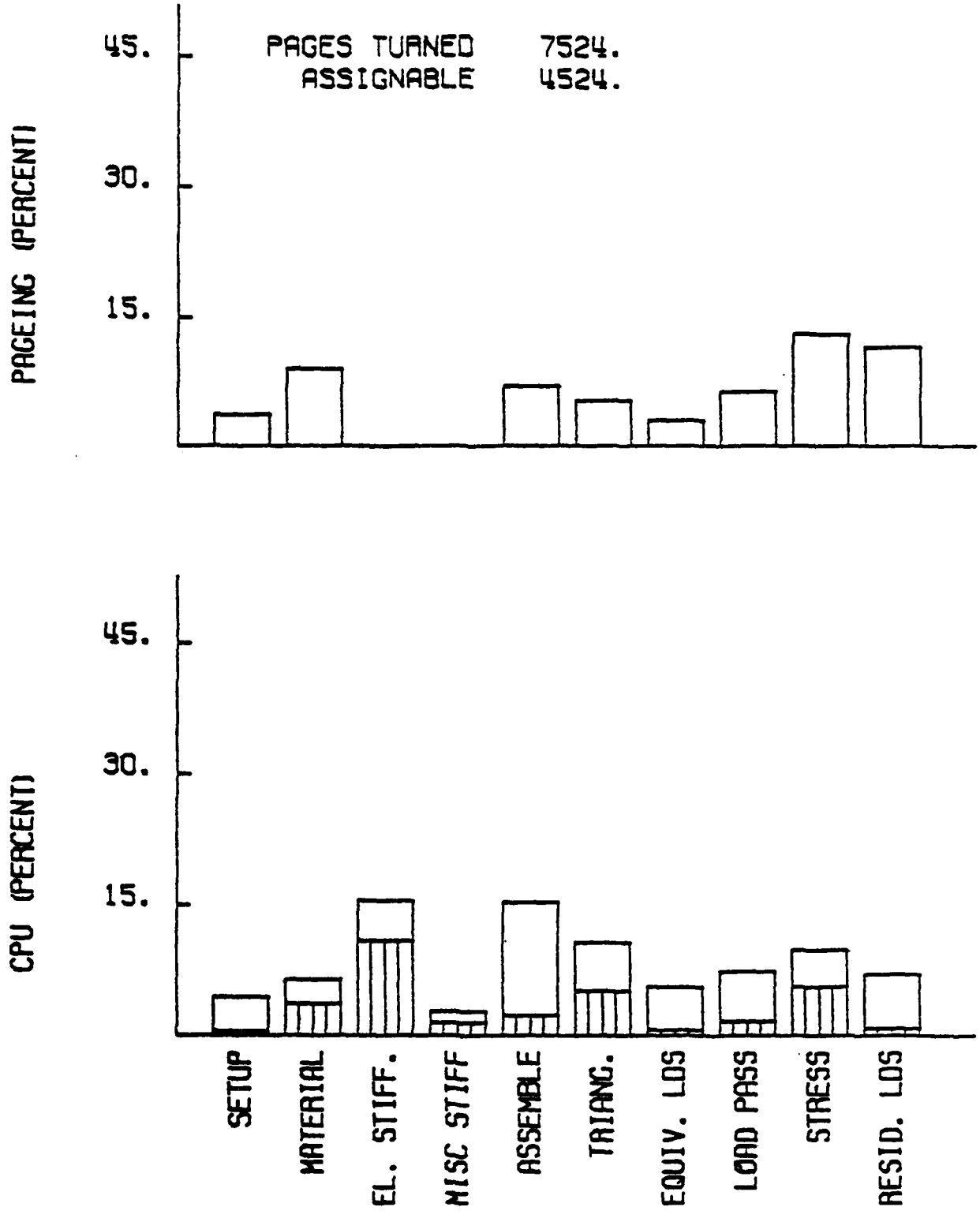
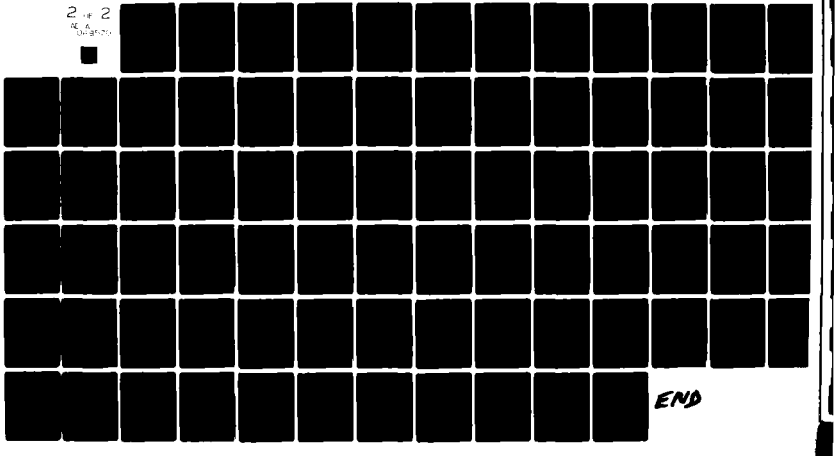


Figure 9. Nonlinear pressure vessel.

AD-A089 570 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/5
OPTIMIZED COMPUTER SYSTEMS FOR AVIONICS APPLICATIONS.(U)
FEB 80 R T CHIEN, L J PETERSON F33615-78-C-1559

UNCLASSIFIED AFAL-TR-79-1235 NL

2 OF 2



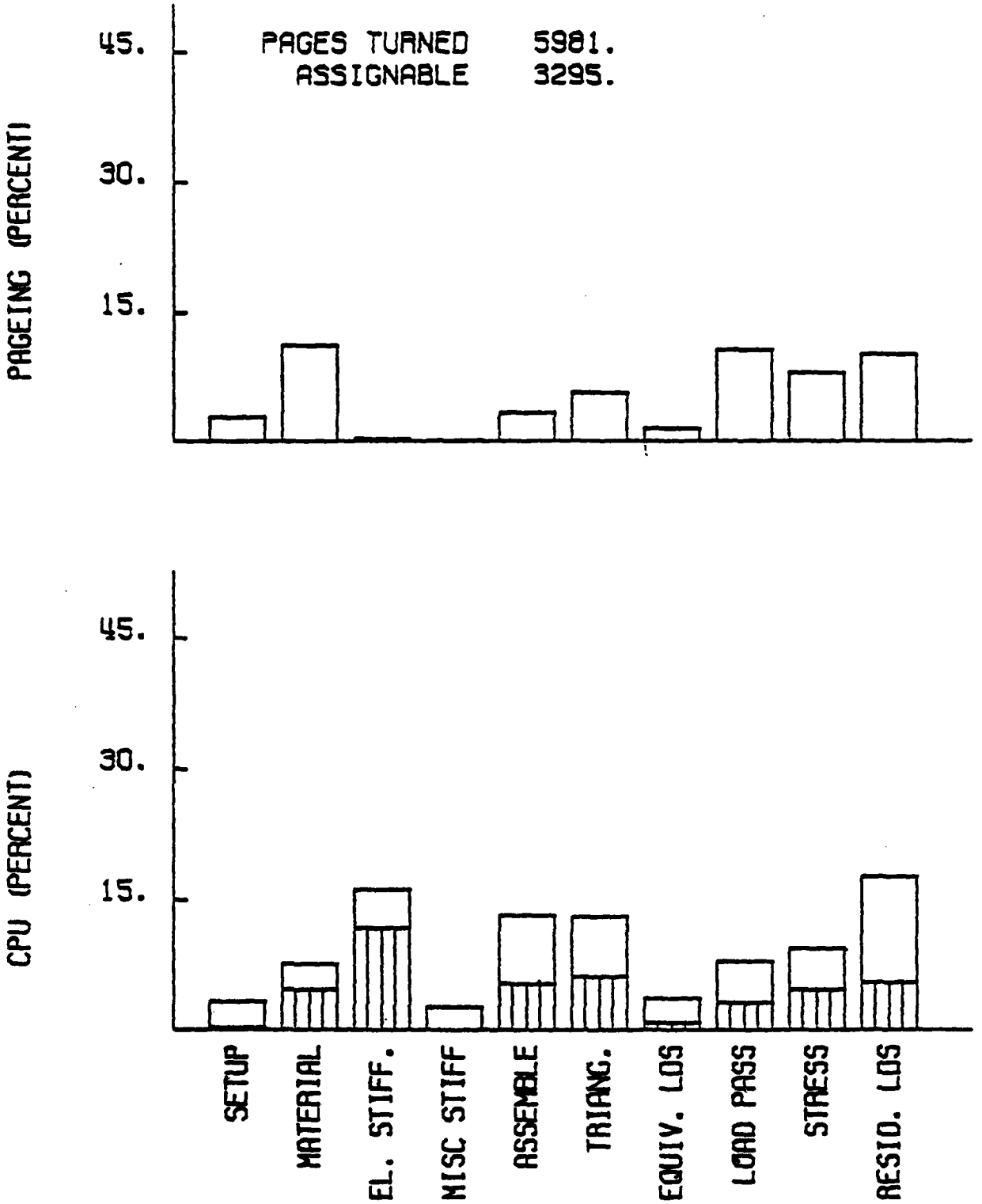


Figure 10. Nonlinear pressure vessel (substructure).

and the pages assignable are indicated. Pages turned are the total number of pages turned during the solution. Pages assignable are those pages turned which were directly related to the movement of data associated with the finite element problem as opposed to paging associated with operating the FINITE system itself. The difference between these two figures drops considerably as the size of the problem increases. Given the percentage of paging and total number of pages turned, etc., one can calculate real clock time associated with paging. These combined with the total CPU utilization permits one to determine the real or clock time required to solve a particular problem. As shown in Figures 8, 9, and 10, CPU utilization is spread through all of the processes associated with finite element analysis, with the element stiffness generation and triangulation standing out as significant.

The second problem solved is the penetrated plate, shown in Figure 11. This problem required a three-dimensional stress analysis. This is a relatively large nonlinear problem. It consists of 664 nodes, 80 elements (20 node cube with five integration points throughout the thickness). The plate has a very broad band with 450 terms. The results are shown in Figures 12 and 13, where it is very clear that element stiffness generation and triangulation of the equations represent the primary utilization of CPU resource. Paging is assignable primarily to the triangulation procedure.

3.4 Analysis of Results

The preceding figures show that for large problems, a significant amount of CPU resource is used for generating the element stiffness, i.e., doing numerical integration and for triangulating the stiffness equations. Certainly an FEM machine must has specialized hardware to aid in these two areas.

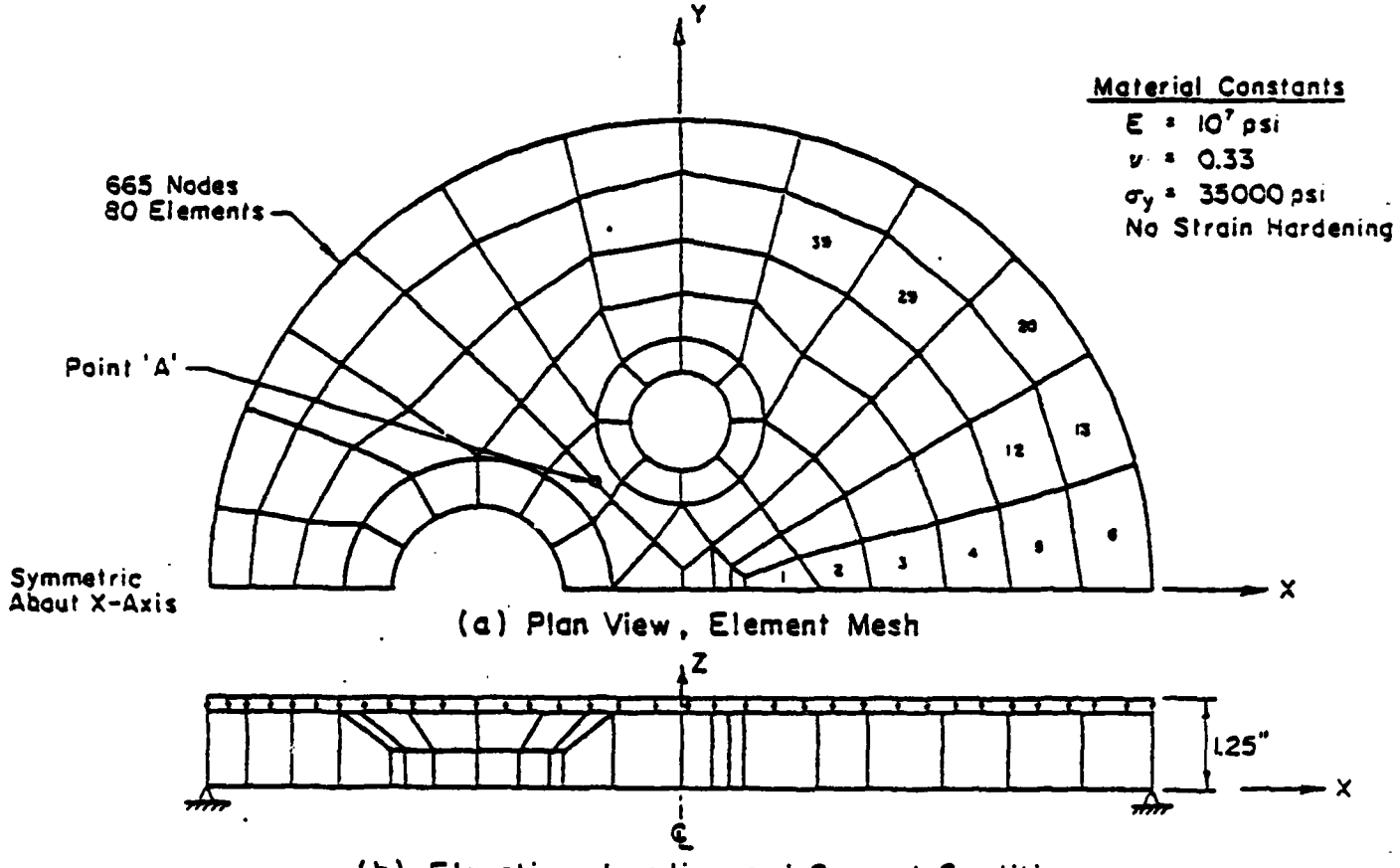


Figure 11. Thick penetrated plate.

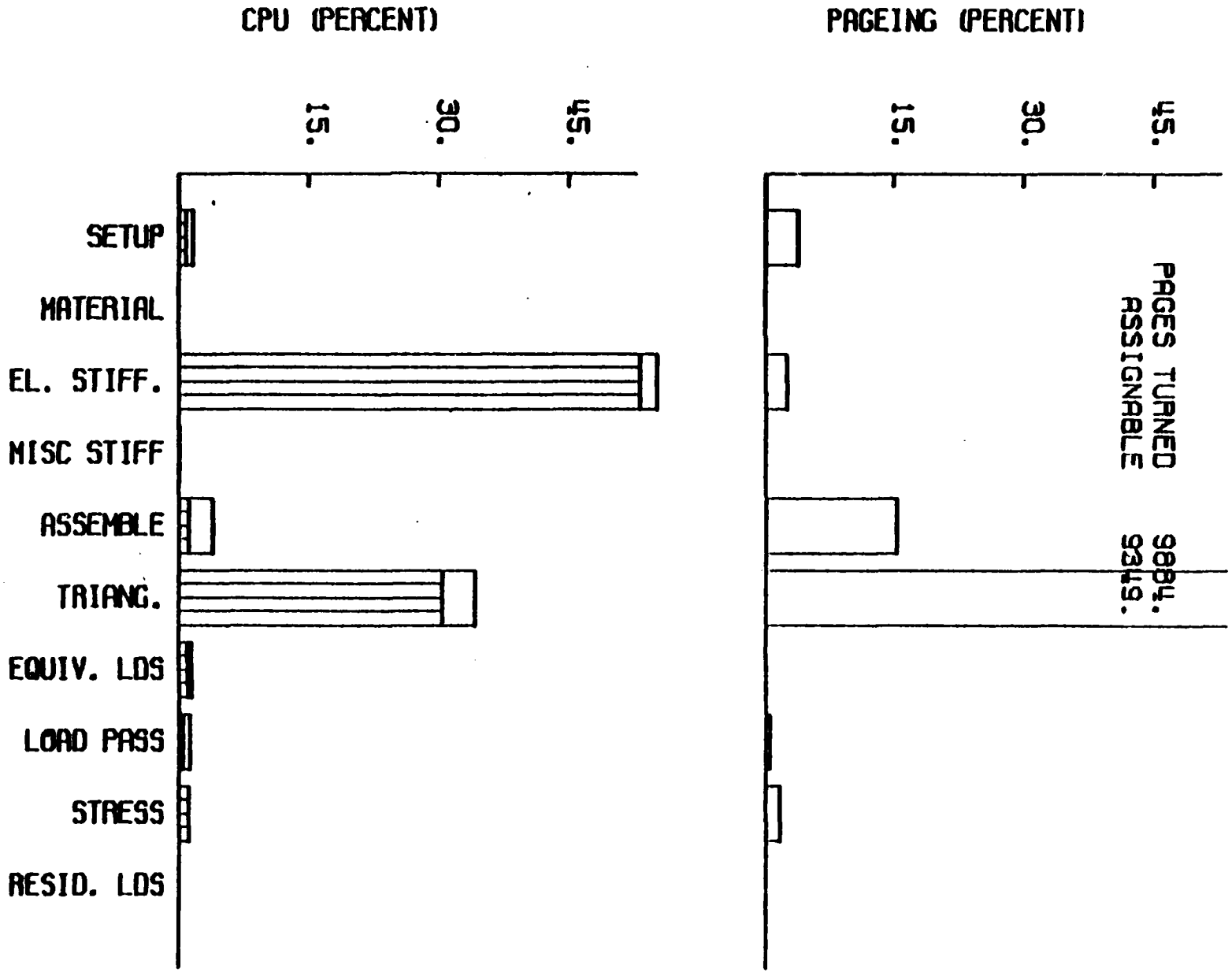


Figure 12. Linear penetrated plate.

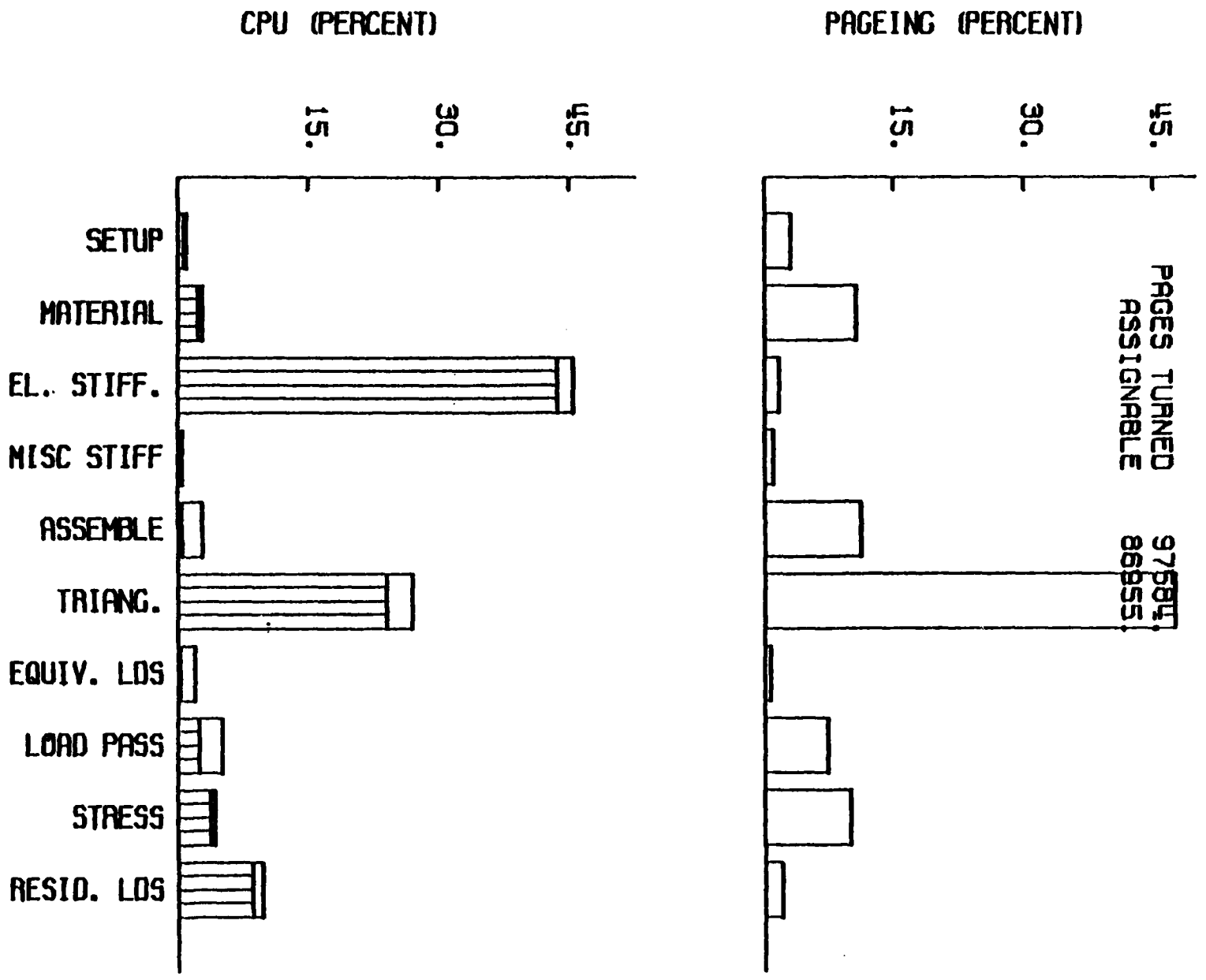


Figure 13. NLPP six load steps (6tr + 20it).

These figures, however, do not tell the entire story. Engineering productivity is not governed by the amount of CPU utilization but rather by the real clock time required to obtain the solution to a problem. CPU utilization reflects only the cost of computation. In order to look at the problem from the point of view of engineering productivity, it is necessary therefore to calculate the wall-clock time required to solve each of these problems. In order to do this it was necessary to assign some figures to the paging speed of the machines to be considered. In this case we used the CDC Cyber/175 and the Burroughs B6700. It was determined that the Cyber/175 can turn approximately 20 pages per second of wall-clock time. Similarly, the Burroughs B6700 can turn approximately twelve pages per wall-clock second. The difference between the two is due to the fact that Burroughs is a virtual computer and does double faulting when the POLO paging system operates within the Burroughs virtual environment. In reality the Burroughs and CDC hardware operate at approximately the same speed. Figures 14 and 15 show the results for the nonlinear vessel and the nonlinear penetrated plate. Note that each bar on the histogram now has three sections. The bottom shows the CPU time, the middle or clear section shows the data management time, and the top crosshatched section shows the paging time. Note that on the Cyber/175, which has a very fast CPU, a large percentage of the time associated with the nonlinear pressure vessel problem is associated with paging wait time. The Burroughs is somewhat more balanced and spends a significant portion of its time utilizing the CPU, the remainder to paging wait time. For the nonlinear penetrated plate, the Cyber does spend a significant amount of time utilizing the CPU. At the same time, the wait time associated with paging during triangulation is significant and does govern the total clock time of the job. That is not true of the Burroughs where most of the time is spent utilizing the very slow CPU. Only a small portion of the time is spent in paging.

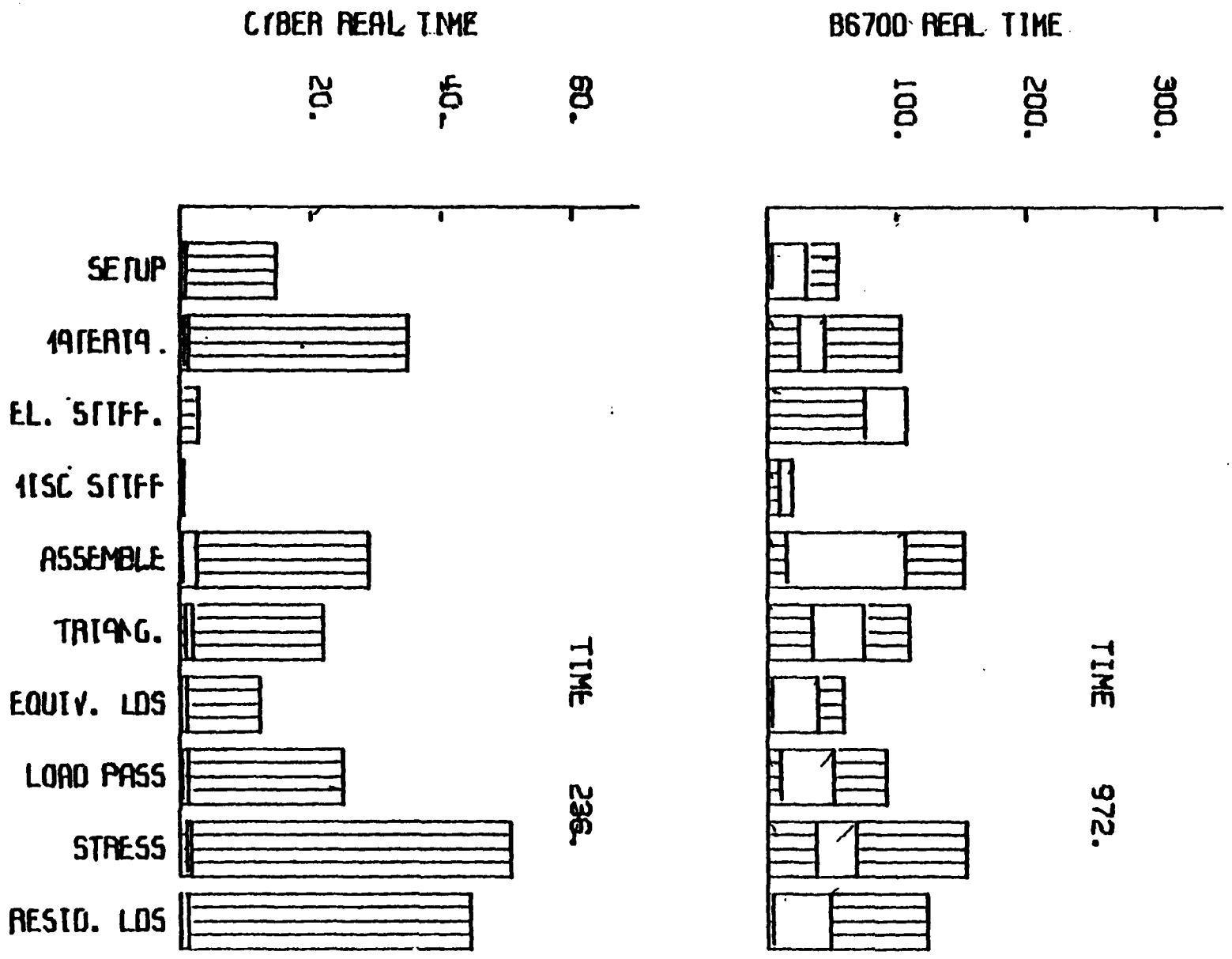


Figure 14. NLPV current machines.

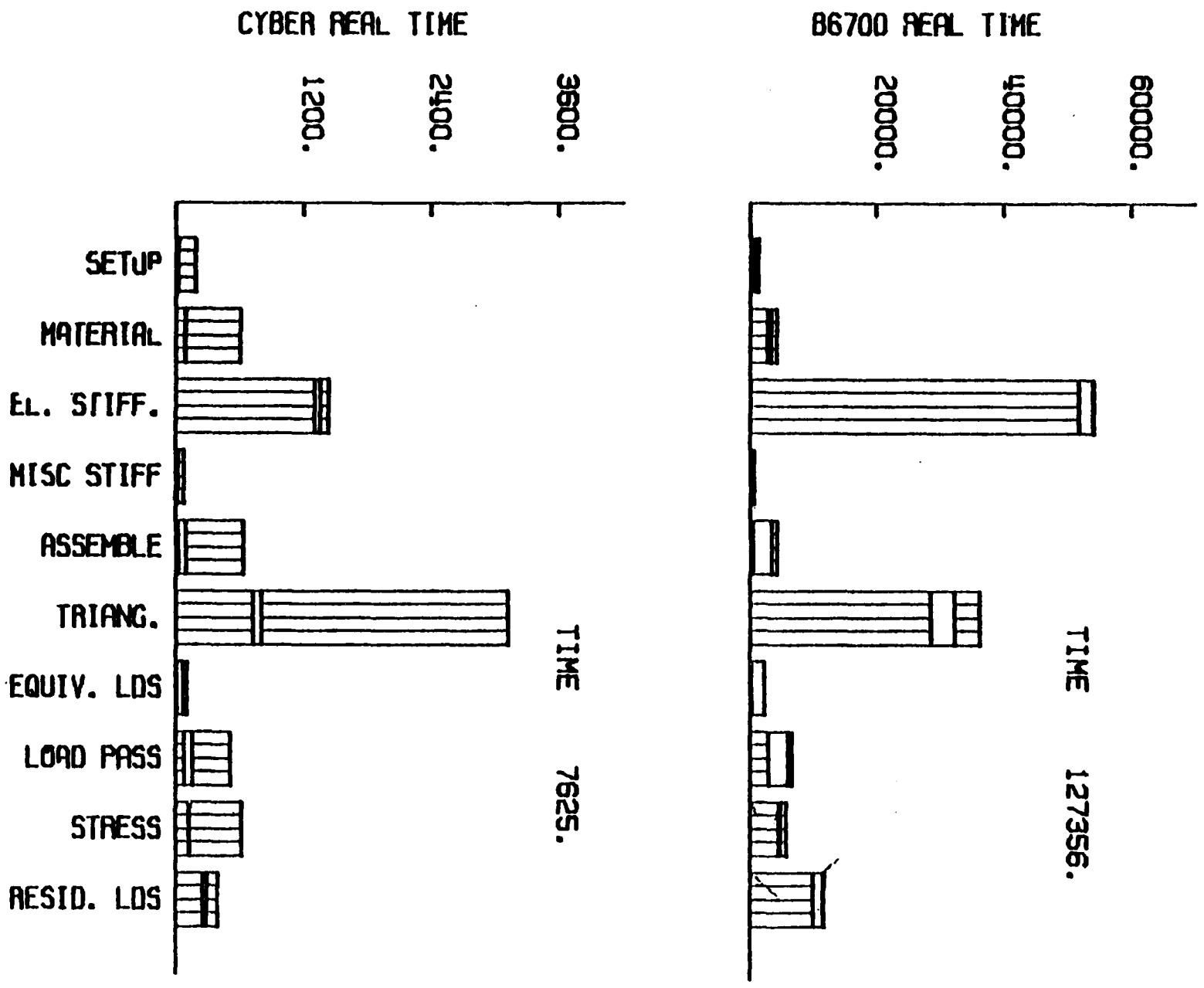


Figure 15. NLPP current machines.

It is interesting now to speculate on the increase in engineering productivity (decrease in clock time required to solve a problem), if the speed of certain functional units within these computers were radically changed. For example, it is possible to either increase the memory of the computers in question significantly, or, as a more cost effective alternative, to increase the speed of the I/O subsystem so that the paging time is reduced significantly. The figures chosen for this example are to increase the speed of the I/O time by a factor of 60. This would correspond roughly to the speed of CCD today. Similarly, one can break the CPU resource utilization into two parts, the data management time, which can be considered scalar CPU utilization, and "number crunching time" such as the times required to solve simultaneous equations or to integrate the stiffness of the individual elements, and which can probably be done in parallel. The parallel operations can be done easily 20 times faster than the current Cyber/175. This number corresponds roughly to the lower bound on the CRAY1 and BSP. Results of these speculative changes are shown in Figures 16 and 17 for both the nonlinear pressure vessel and the nonlinear penetrated plate, for both the Burroughs B6700 and the CDC Cyber/175. As might be expected, changing the speed of the I/O device significantly improves the performance of the Cyber/175, while changing the speed of the processor significantly improves the Burroughs B6700. Changing both cuts the real time between 87 and 90 percent on the penetrated plate and between 62 and 95 percent on the nonlinear pressure vessel. One might note that changing from a Burroughs B6700 to a VAX 11/780, a machine with a more realistic "slow" processing speed for today's standards, would make the figures on the slower machine significantly better. The authors did not have access to a VAX, and the FINITE system was not running on one. Therefore, these figures were not generated.

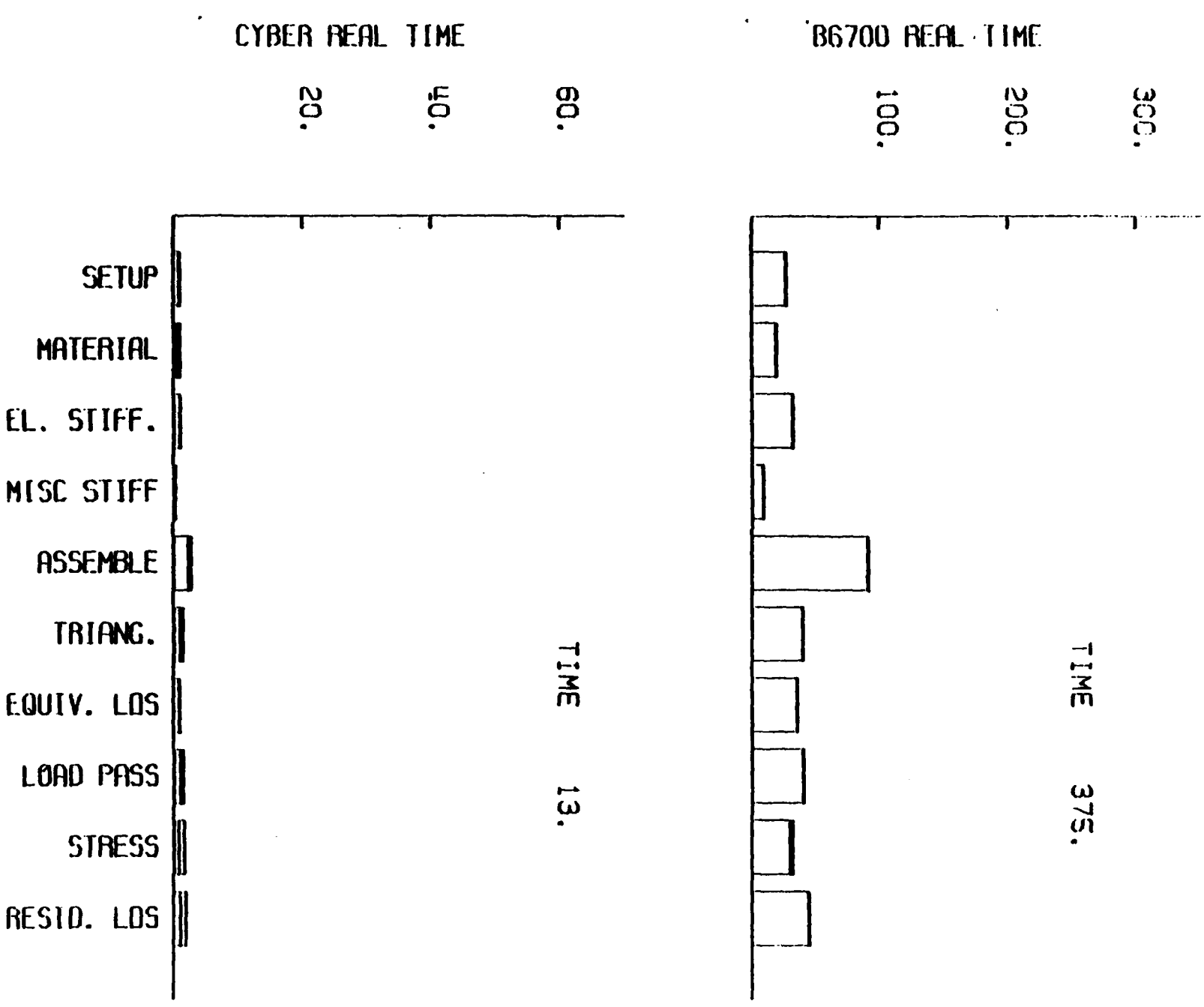


Figure 16. NLPV fast array and I/O.

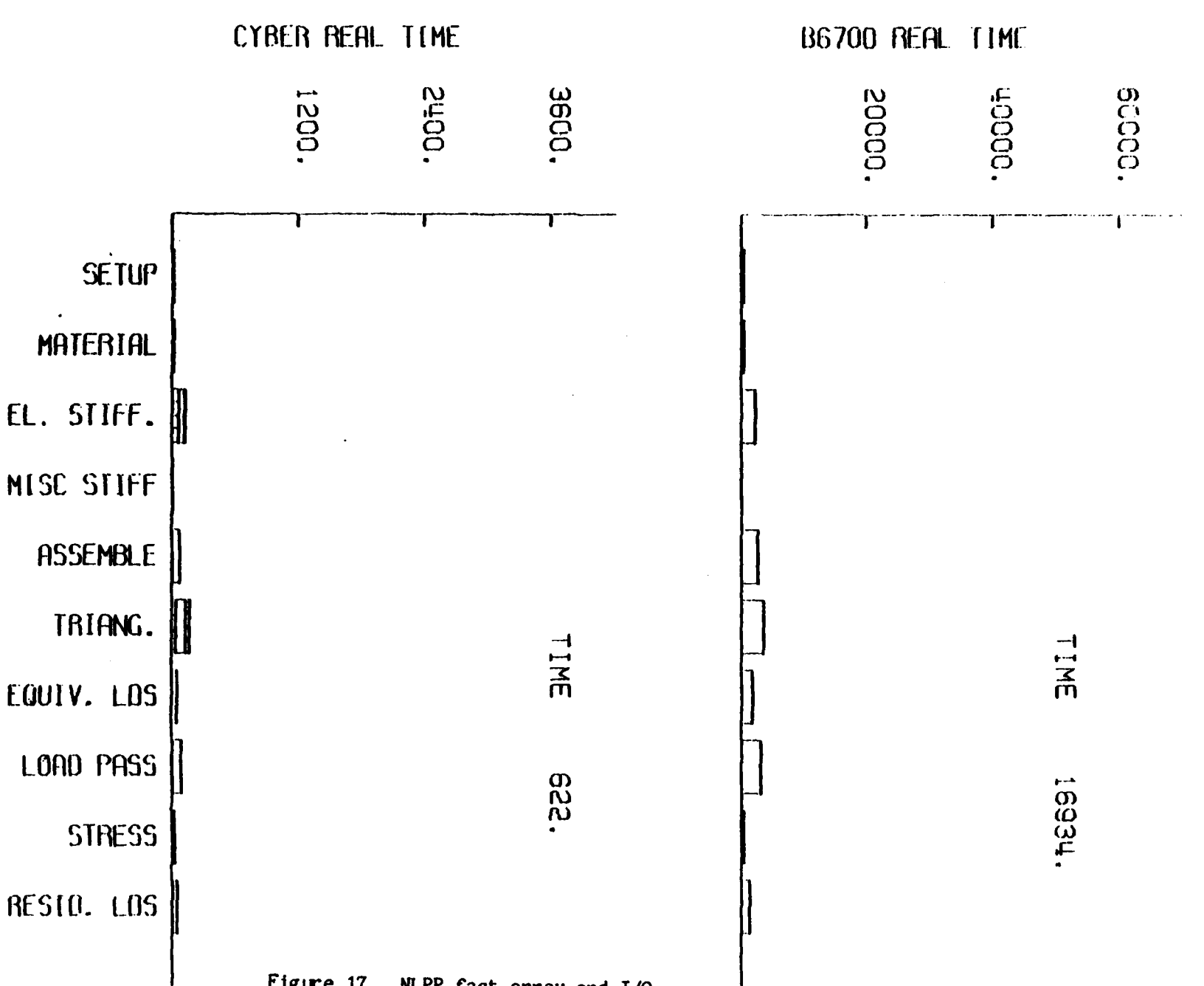


Figure 17. NLPP fast array and I/O.

The conclusions to be reached from the histograms and from generated through speculative changes in hardware, are that s improvements in engineering productivity can be achieved by t hardware toward the finite element process. Finding a fast CPU th type of parallelism, array processor, multi-processor, lock step step parallel processor, etc., will solve part of the problem. Par it will reduce the times required to obtain a solution on small ma permit possibly the introduction of a so-called finite element ma the average office. Additional and significant increases in real are also to be obtained through some type of extended memory, n memory, or fast paging device. The actual levels of improvemen will, of course, depend on the speed of the base machine.

3.5 Data Mapping Hardware

One characteristic of the finite element method is that it re to handle large numbers of small matrices, if one writes the softwar the logical form of the direct stiffness method. For example, the matrix of the 20-node isoparametric element used in the soluti nonlinear penetrated plate is logically represented by 192 3 X 3 The maintenance of these matrices via DBMS can be quite slow. I some systems today generate the stiffness of the entire element mat large block; in effect, the software operates on a non-logical bas to gain speed. It would seem that machines of the future would not engineering system designer into this unnatural mode of operation this has not been done in the FINITE system. The result is th processes have a significant amount of overhead associated with the apparent when one examines the histogram associated with the line vessel. This problem had a large number of elements. The time

generate the stiffness of each element is small. The amount of data management time is then quite large during element stiffness generation and during assembly due to the fact that the system must handle these large volumes of small matrices. This problem is aggravated when one solves aerospace type problems which consist of large numbers of frame and beam elements. These are very simple elements requiring virtually no CPU utilization to generate the stiffness matrix. Therefore, all of the time associated with stiffness generation and assembly is really attributable to manipulation of the large numbers of elements (these problems often have one to 2000 components). Thus it would appear in considering an architecture for the finite element analysis, that it is necessary to consider not only a fast CPU and a fast "paging" system, but also a system which can very rapidly handle large numbers of small sparse matrices. A machine which possesses these attributes would definitely be a significant improvement over what is available today, especially if it would be in the $200,000 - $300,000 range, since that would permit it to be used in a large number of engineering installations.

The preceding sections have been used to show the character of the finite element problem and how machine resources are utilized during the solution of some of today's simpler problems. The team at CSL has been studying the kernels of the algorithms for solving the simultaneous equations associated with the finite element method. Various machine architectures are being studied for problems such as memory conflict and speed of computation. The integration kernels are also being examined with preliminary emphasis on the dominant matrix multiplication subkernel. Hopefully, these studies will lead to some conclusions concerning necessary computer architecture.

When these studies have been completed, the problems of data mapping and paging will be studied to form a complete finite element machine architecture.

3.6 Computer System Evaluation

Preliminary evaluation has focused on the two critical kernels of code for finite element methods: matrix multiplication and solving linear equations. Known algorithms for linear equation solving have been collected. Some existing evaluations of these two kernels are available in the literature as well. Existing evaluative studies are concerned with medium scale computers, specialized array processors, and large scale vector-oriented computers. Little if any work has been done by other groups on multiple processor approaches to solving these kernels. Furthermore, existing studies exhibit a noticeable lack of information regarding comparison of the effectiveness of widely varying alternative computer architectures and algorithms. We therefore felt that our pilot study should focus primarily on a preliminary evaluation of alternative algorithms for a multiple processor approach to the kernels. We further felt that a serious comparative evaluation of alternative computer architectures should be postponed until more is known about multiple processor approaches. Likewise, the evaluation of alternative memory hierarchy organizations was not emphasized as yet since that work could best be done in the light of known requirements of preferred architectures for the computation aspects of the problem.

The primary vehicle used for experimental multiple processor studies was the AMP-1 multiple microprocessor system located at the Coordinated Science Laboratory. This system was constructed, partially with the support of the Joint Services Electronics Program, for the purpose of evaluating a variety of multiple microprocessor configurations with shared memory, studying job

multitasking for parallel execution, and evaluating effective organizations for achieving high concurrency of computation and reducing memory access contention. A block diagram of the AMP-1 system appears in Figure 18.

This system employs eight Motorola 6800 microprocessors which access memory over a shared bus system using a strict round-robin bus window access discipline. The memory is organized as 64 modules of 1K bytes each. Software for this system is written in Motorola assembly language. Concurrent Pascal is also available and is executed by an interpreter. None of the software written for this system depends on a specific number of processors. Rather, it is subdivided into a large number of independent tasks placed in a common job queue. Scheduling is accomplished simply by having an idle processor interrogate the queue for its next job. The memory mapper allows each processor to have a small amount of logically local memory. Local memory is used as temporary working storage and to store enough of the processor state to permit convenient reentrant programming so that the processors can share the same code. The BBX interface connects this system to a DEC System 10 computer. The System 10 can read and write any location in memory and can start, stop, and reset arbitrary combinations of processors. It can also be interrupted for message passing from the processors. The special locations are designed to provide certain functions not otherwise obtainable in the Motorola 6800 processor such as: support for critical sections, interrupting and transmitting status to the DEC 10, and interacting with the memory protection subsystem (not shown). These locations are accessed as if they were memory locations. When operating at full speed, the memory modules have a cycle time of 5, i.e. when a processor accesses a memory module the next 4 processors in sequence are forbidden from accessing that memory module. The busy checker determines whether a memory access request is attempting to access a busy module and if so, disables the clock for that processor for one

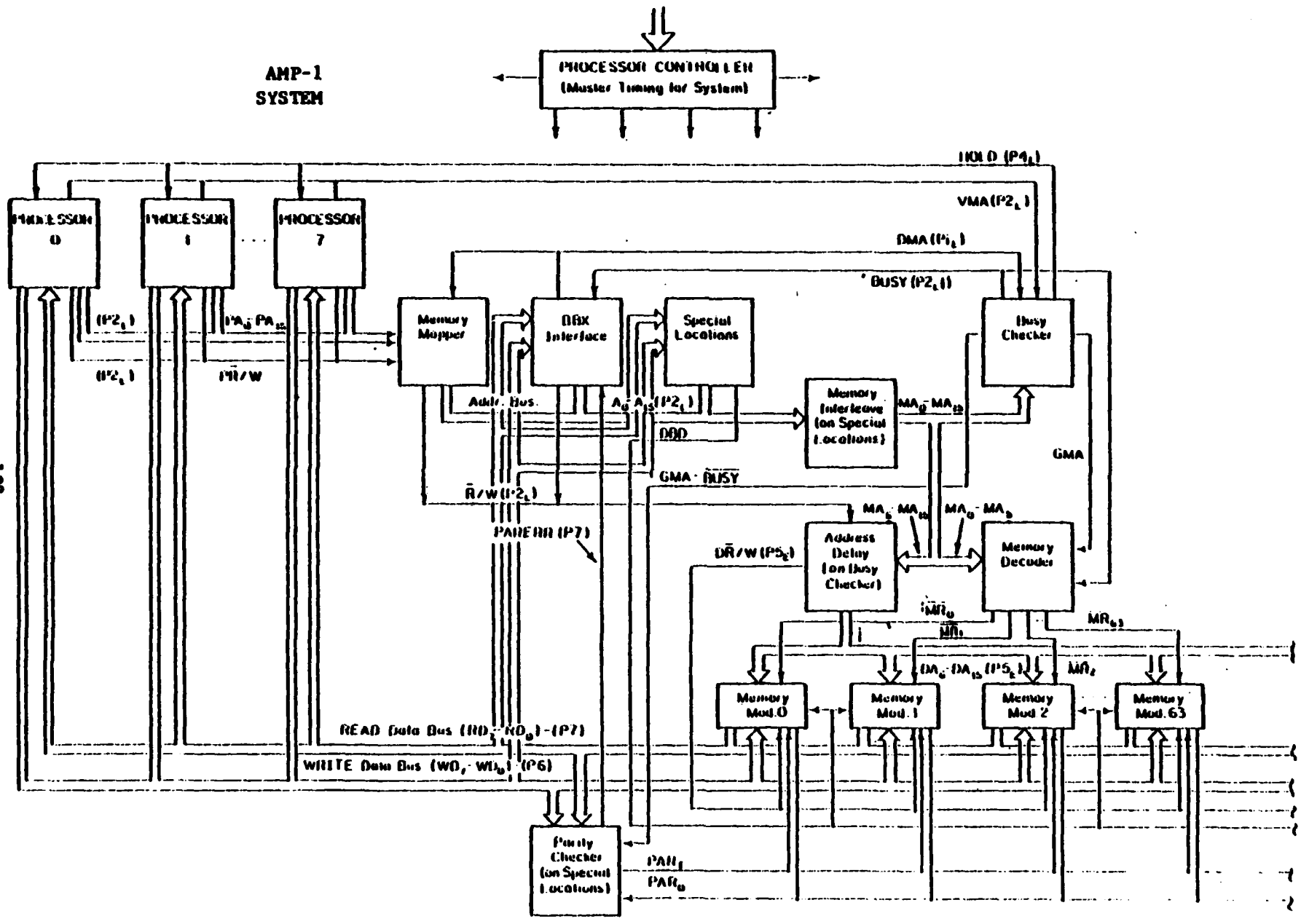


Figure 18. The AMP-1 system.

complete cycle. The design of the Motorola 6800 processor and the clock disable logic permit a rejected memory access request to be resubmitted on the next cycle. Special logic has been constructed to guarantee that each processor can get access to the memory within 10 microseconds. This guarantee is necessary due to the fact that the Motorola 6800 processor is implemented with dynamic logic.

Several design decisions for this system were made to permit the system to be used for serious experimental study of the memory access conflict problem of multiple processor shared memory systems. The Motorola 6800 was selected as the processor in part due to its intensive use of memory. Several alternative processors access memory far less frequently and sporadically. Arbitrary combinations of processors can be used in a particular experiment. No software modification is required to select a different combination of processors. A memory interleaving plug is provided to allow an arbitrary selection of the address bits as the memory module number. This feature allows the study of uninterleaved address, by selecting the top 6 bits, up to fully interleaved memory by selecting the low order 6 bits. Other plugs can select 2-way, 4-way, 8-way, 16-way, and 32-way, interleaving. The busy checker permits an extension of the memory cycle time beyond 5 clock times to 6, 7, or 8 clock times. Furthermore, it also allows a reduction in the number of memory modules below the 64 implemented in the system. Thirty-two module operation is provided by forming pairs of the 64 modules and assuming that both module 0 and module 1 are busy whenever either is busy. Similarly, 16, 8, 4, 2, and 1 module systems can be emulated.

The AMP-1 multiple microprocessor system has a potential performance of 8 times that of a single Motorola 6800 processor. Experimental studies with this system should be oriented toward discovering how nearly the actual performance obtained from this system approximates the performance capability

of the system and what effects are responsible for the degradations from ideal performance. Used in this fashion, the AMP-1 system can be an important experimental tool for exploring the fundamental properties of multiple processor systems and a means for evolving effective multiple processor systems of the future. Preliminary experimental studies of matrix multiplication and Gaussian elimination (a classical technique for solving linear equations) were performed using the AMP-1 system. A special hardware monitor was constructed to measure the performance of the system. Details of these studies are presented in the following subsections.

3.7 Matrix Multiplication Results

A series of experiments were performed for evaluating matrix multiplication performance on the AMP-1 system. A program for matrix multiplication was written in Motorola 6800 assembly language. Memory banks 0 and 1 were used as private memory for the processors, the main program is stored in bank 2, the code for the dot product routine is stored in bank 3, the floating point multiply routine is stored in bank 4, and normalization and add routines are stored in bank 5. For the matrices themselves, a 5 byte floating point format is used for each matrix element. This format provides for an 8 bit exponent and a 32 bit mantissa which allows numerical precision comparable to most large computers. The matrices used are 32 X 32 in size. Thus each matrix requires 1024 5 byte locations in memory. The matrices thus occupy logical banks 6 through 20, i.e. 15K storage locations. The computation to be performed is A X B = C, where A, B, and C are 32 X 32 matrices. The system itself uses some special locations in logical bank 63. The logical bank numbers used in this discussion are simply a reflection of the address space as seen by the programmer, i.e. the logical bank number may be decoded from the high order 6 bits of an address used in the program.

Logical banks correspond to actual memory modules when 64 memory modules are used and no interleaving is used. Use of an interleaving plug to provide 2 to 64-way interleaving will alter the placement of logical addresses among the memory modules in a way which is transparent to the programmer. In these preliminary experiments, the number of banks was always 64, regardless of the degree of interleaving. Thus the performance for no interleaving or a low degree of interleaving as indicated is higher than one would expect if the number of banks were equal to the degree of interleaving. The matrix multiplication is divided into 1024 separate Dot Product jobs which are independent and separately scheduled.

The time required for a floating point addition, with the format used, averages 305 cycles. Some variance in the add time occurs due to the variable number of shifts required for normalization. The multiplication routine requires an average of 4,669 cycles. A Booth algorithm is used which performs an add followed by a shift for a 1 bit in the multiplier and a shift only for a 0 bit. The Dot Product requires 159,436 cycles. The Dot Product job requires 32 adds and 32 multiplies which accounts for 159,168 of these cycles. Thus 99.83% of the time spent in a Dot Product job occurs within the floating point add and multiply routines. The times above were measured in actual performance with one processor executing the program. Thus, no memory conflict cycles were present. The entire matrix multiplication job as coded in our program, MXMC, with one processor requires 163,309,473 cycles. The 1024 Dot Product jobs account for 163,262,464 of these cycles. Thus 99.97% of the time is spent in the Dot Product routine. This indicates an extremely low period of time required for processor scheduling. The 32,768 additions and 32,768 multiplications require 162,988,032 cycles. Thus 99.80% of the time for the MXMC program is spent in the floating point addition and multiplication routines. Real time for the Motorola 6800 microprocessors is

one microsecond per cycle. These results indicate that the Motorola 6800 microprocessor is not suitable for high precision floating point operation. One or two orders of magnitude in performance of these floating point routines could be achieved by using a faster microprocessor, such as the AMD2900 series microprocessor with a wider word. Times can also be improved by using specialized pipelined floating point units. Further speedups could be obtained by using more processors or function units in the configuration of the system.

Nevertheless, performance data taken as a function of the number of processors used, p, and the degree of interleaving, I, indicates the effectiveness of such a multiprocessor approach with shared code and shared memory. These data are shown in Figure 19. When p processors are used, each time a processor accesses memory, the following $\frac{p-1}{2}$ processors are locked out from that module on the average. The original program and data are spread over 22 logical banks of memory, resulting in the use of 22 modules when I = 1, 44 modules when I = 2, and the full set of 64 modules when I is greater than or equal to 4. It is interesting to note that even though so many modules of memory are used with respect to the number of processors a high degree of memory address interleaving among these modules is also required for high performance. Also shown on the curves of Figure 19 is the ideal performance, namely a speedup of a factor of p when p processors are used, relative to the performance for a single processor. High degrees of interleaving result in a performance which comes remarkably close to the ideal, despite the memory access contention due to shared code as well as shared data space for the matrices A and B. Further experiments will be performed to determine performance when the number of memory banks is equal to the degree of interleaving. Also performance as a function of the relative speed of the memory vs. that of the processors will be determined by running

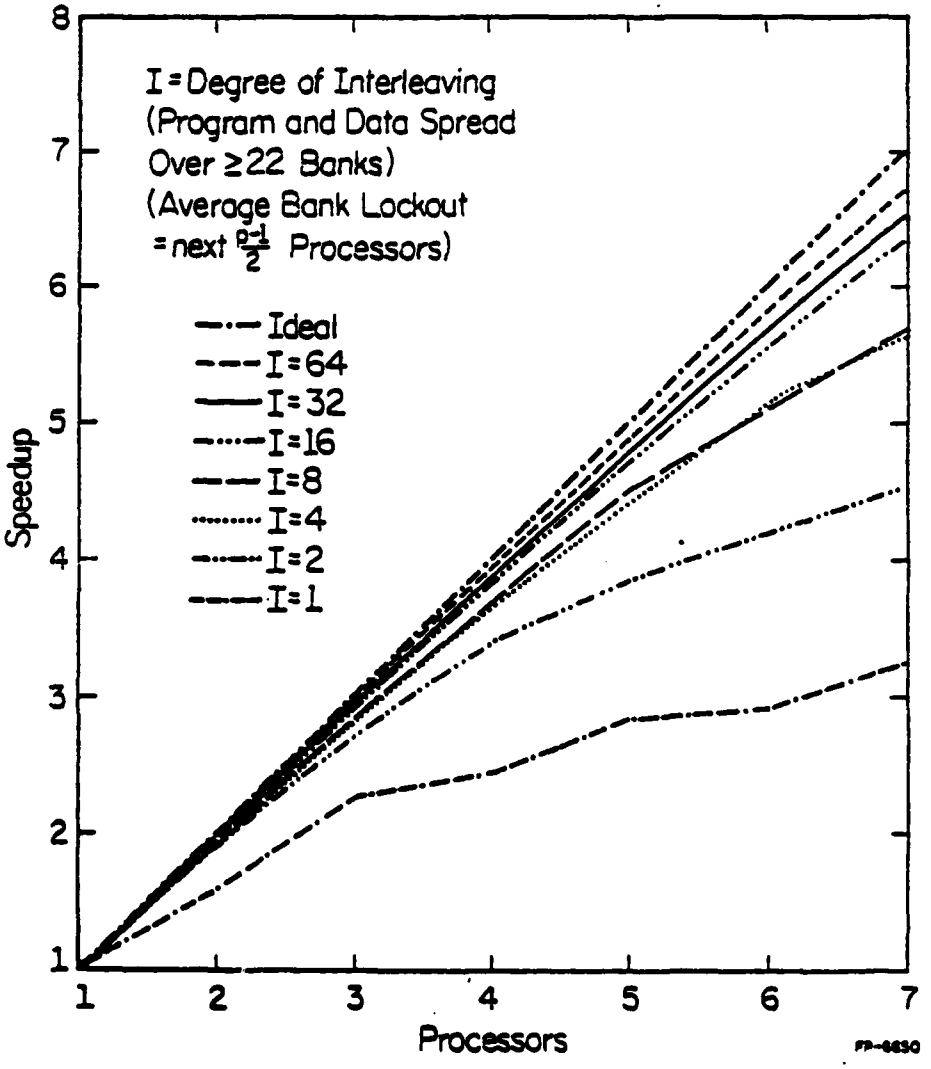


Figure 19. MXMC performance.

future experiments with slower memory.

To aid in the interpretation of these results, a memory access conflict model has been developed for shared memory multiprocessor systems. An outline of this model is shown in Figure 20. The model assumes that each memory access request has an equal probability of referencing any module in the memory. Furthermore, for each processor cycle, that cycle is assigned to be one of three kinds: an internal cycle making no memory reference at all, a cycle involving a memory access request which is accepted, or a cycle involving a memory access request which is rejected. It is assumed that once a rejected memory request cycle occurs for a particular processor, successive cycles for that processor will be rejected memory request cycles until an accepted memory request cycle occurs. Rejected memory request cycles are referred to as conflict cycles. Cycles which are either internal or accepted memory request cycles are referred to as non-conflict cycles. Thus, can be measured for an actual program as the number of accepted memory request cycles divided by the total number of non-conflict cycles. This was measured as 79.17% for the program MXMC. Given knowledge of Y, the number of memory banks, M, and the effective memory cycle time, c_e, the model predicts the probability that a request will be accepted and determines the number of rejected memory request cycles to be added. A Markov modeling approach is used to derive the formula for the probability of acceptance. Internal cycles are assumed to be independently distributed with respect to memory reference cycles. While Y is the request rate to memory from a processor in a non-conflict situation, Y_M is that probability for referencing a particular bank. Thus Y_M is equal to Y divided by the number of banks. Furthermore, is the actual request rate seen by the memory in a conflict situation, i.e. reflects the memory access requests which are rejected by the memory. Finally, Y is the average total number of cycles of actual run time required

MEMORY ACCESS CONFLICT MODEL

(FOR BANKS = DEGREE OF INTERLEAVING = M)

ψ = Probability that non-conflict cycle references memory = .7917 for MXMC

ψ_M = ψ for one bank = .7917/64 for MXMC

$c_e - 1$ = average number of successive processors locked out of a bank (if requested) once access granted = (p-1)/2 for AMP

P_A = Probability that an actual memory access request is accepted

$$= \frac{1}{1+\alpha(c_e-1)} \quad \text{, where } \alpha = \frac{1}{P_A(\frac{1}{\psi_M}-1)+1}$$

$$= \frac{-c_e + (\frac{1}{\psi_M}-1) + ((c_e - (\frac{1}{\psi_M}-1))^2 + 4(\frac{1}{\psi_M}-1))^{\frac{1}{2}}}{2(\frac{1}{\psi_M}-1)}$$

$$\text{Speedup} = \frac{\text{run time for 1 processor}}{\text{run time for p processors with conflict}}$$

Speedup (modeled) = p/ρ, where $\rho = (1-\psi) + \psi\,(1/P_A)$

1

Figure 20. Memory access conflict model for MXMC.

for each cycle of the program if no conflicts were present. Thus Y is computed by counting one cycle for each internal cycle and $1/P_A$ cycles for each accepted memory reference cycle, where P_A is the probability that an actual memory access request is accepted.

Figure 21 shows the extremely close correspondence between the modeled probability of acceptance and an actually measured probability of acceptance as well as the similarly close correspondence between the modeled speedup and the measured speedup as a function of the number of processors with 64-way interleaving. While further experiments are yet to be done, these preliminary results justify some confidence that the model is fairly accurate at predicting the amount of memory access conflict for the MXMC program and that at least for a high degree of interleaving the difference between the actual speedup obtained and the ideal is accounted for by memory access conflict. We may infer from such differences as do exist between the model and actual performance measurements that the measurements show slightly higher performance for low p due to the sequential access effects of real programs which increase performance relative to the uniform access assumption of the model. These effects become insignificant when a larger number of processors are run due to the interleaving of their access request streams which has a randomizing effect. For larger numbers of processors, the measured performance is somewhat lower than that predicted by the model due to performance degradation effects other than memory access conflict, e.g., the presence of critical sections of code such as job scheduling for MXMC and job precedence constraints in general which cause some processors to wait while other processors continue with their computations. While these differences are insignificant for the particular experiments run for matrix multiplication they should be expected to become more significant for other cases and other problems.

(64 - WAY INTERLEAVING)

P	MODEL P_A	MEASURED P_A
1	1	1
2	.9938	.9955
3	.9876	.9877
4	.9814	.9813
5	.9753	.9795
6	.9691	.9706
7	.9629	-
8	.9527	-

P	MODEL SPEEDUP	MEASURED SPEEDUP
1	1	1
2	1.9902	1.9913
3	2.9705	2.9657
4	3.9409	3.9355
5	4.9017	4.8927
6	5.8523	5.8183
7	6.7928	6.7435
8	7.7233	-

Figure 21. Model for MXMC.

Preliminary conclusions reached from this study are that sha multiprocessors can achieve low access conflict and efficient job s Further evaluation is needed for other architectures and the effect matrices which require paging from secondary memory. Several algo known for matrix multiplication which have been tailored to speci oriented machines and the paging problem. These algorithms will useful starting point for the continuation of this research. Fi success of our memory access conflict model in the case studied just further work on this model be done to extend it to modeling conf general resources including not only memory but shared function critical sections of code. Such a generalization will prove to be for quick evaluation of wide ranging alternative architectures exec complicated jobs.

3.8 Analysis of Gaussian Elimination

A program was developed for the AMP-1 system to perform elimination for a 14 variable problem. This program uses the s floating point format described previously for matrix multipli solves the equations Ax = B, where A is a 14 X 14 matrix and B i column. The program itself was adapted from a FORTRAN coded progra extensive evaluation had been done on various IBM System/360 particularly a broad class of machines similar in nature to t pipelined Model 91. The version coded for the AMP-1 uses logical b for private memory, banks 2 and 3 for program memory, bank 4 for th point multiply and divide routines, banks 5 and 6 for the A and which total 1050 bytes, and bank 63 for reserved locations for t Flags used for insuring that proper precedence among jobs is guar stored in bank 62 for the original version of the program (GAUSB) a

57 and 58 for two other versions of the program (GAUSY and GAUSZ).

The program is divided into three kinds of jobs: normalize, N_I, row reduction, R_{IJ}, and back substitution, B_J as described in Figure 22. Each normalize job sets a diagonal element of the A matrix to 1. It assumes that the elements before the diagonal element are all 0 and divides the elements after the diagonal including the B column element by the prior value of the diagonal element. A row reduction job, R_{IJ}, sets one element to the left of the diagonal in row I to 0 by replacing row I with an appropriate linear combination of row I and row J. It assumes that the diagonal element of row J has already been normalized to 1. After all normalization and row reduction jobs have been completed, the A matrix contains 1's along the diagonal and 0's below the diagonal. At this point the back substitution jobs may begin. A back substitution job, B_J, adjusts the B column elements above row J to values they would attain if the elements above the diagonal of matrix A in column J were reduced to 0 by replacing rows with linear combinations of the corresponding row and row J. A great deal of dependency thus exists between the various back substitution jobs. Job precedence flags must guarantee that before an element of the B column is adjusted by job B_J it has been adjusted by all back substitution jobs whose subscripts are greater than J. The precedence between these jobs is shown graphically in Figure 23. There are a total of n normalized jobs, $n^2 - \frac{n}{2}$ row reduction jobs, and n - 1 back substitution jobs. The number of jobs which can be performed in parallel on a multiprocessor system thus varies anywhere from 1 to n - 1. Job scheduling can thus be a critical component in determining the performance of a multiprocessor system. The actual numbers of floating point operations performed by each job of the Gaussian elimination program are show in Figure 24.

N_i : Normalize row i = set $A_{ii} = 1$

(divide row i by A_{ii})

(assume $A_{ij} = 0$, i.e. R_{ij} completed, $\forall j < i$).

R_{ij} : Reduce row i by row j = set $A_{ij} = 0$

(j<i) (replace row i with row i - A_{ij} × row j)

(assumes $A_{jj} = 1$, i.e. N_j completed).

B_j : Back substitute Column j = $A_{ij} = 0$ $\forall i < j$

(replace B_i with $B_i - A_{ij} \times B_j \forall i < j$)

(assumes all N_i, R_{ij} jobs completed

and B_i adjusted for B_k,

i.e. B_k completed through row i, $\forall k > j$.

Figure 22. Gaussian elimination jobs.

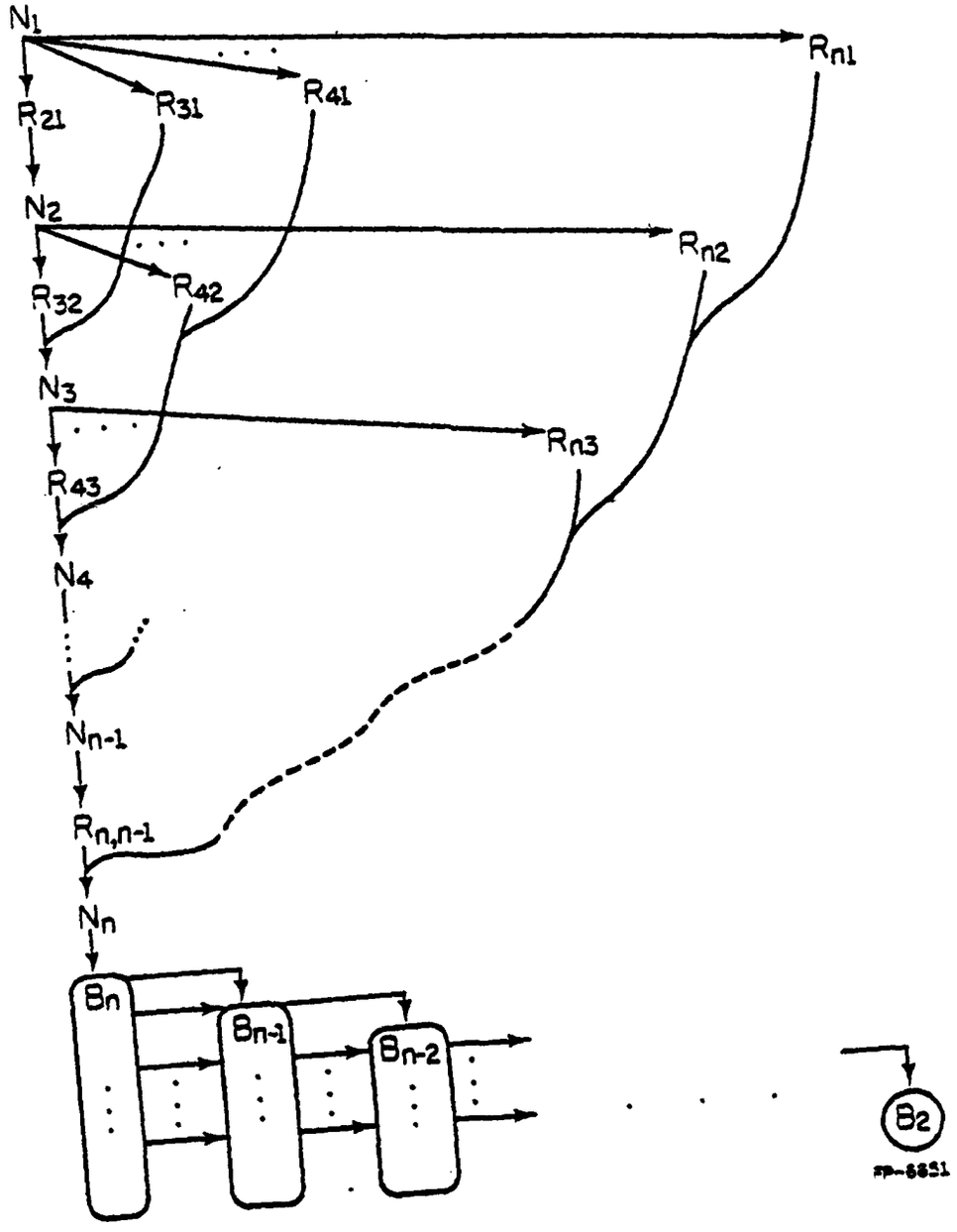


Figure 23. Gaussian elimination job precedence.

JOB COMPLEXITY (A is n x n)

N_i:	$n - i + 1$	divides	$(1 \leq i \leq n)$
R_{ij}:	$n - k + 1$	multiplies	$(1 \leq j < i \leq n)$
	$n - j + 1$	subtracts	
B_{ij}:	$j - 1$	multiplies	$(2 \leq j \leq n)$
	$j - 1$	subtracts	

TOTAL OPERATIONS:

$\frac{1}{2}n^2 + \frac{1}{2}n$		divides
$\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$		multiplies
$\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$		subtracts
$\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$		flops

Figure 24. Job complexity when A is n X n.

The performance of two extreme computer architectures executing a Gaussian elimination program can be expressed in terms of these figures. For illustration the number of floating point operations (flops) which must be performed in series is used as an indication of run time. For a serial machine, floating point operations are performed one at a time. Thus the run time is approximately proportional to the total number of floating point operations required, namely $\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$. At the opposite extreme, a data flow machine may be envisioned which has sufficient parallelism so that any greater degree of parallelism would result in no performance improvement at all. For such a machine the row reduction jobs shown on a single row in Figure 23 can all be performed in parallel. It is convenient to refer to this combination as a single job, R_j. Similarly, to simplify the analysis, it is convenient to recombine the back substitution jobs shown in Figure 23 so that each row of Figure 23 forms a new back substitution job. Thus the task of the new back substitution job is to perform all adjustments on a particular element of the B column. The job precedence then requires alternating N and R jobs until they are all complete and then performing the B jobs one at a time. One floating point operation (flop time) is required for each normalize job. The number of actual operations performed in parallel in a particular one of these jobs varies from 1 to n. For each of the n - 1 row reduction jobs two flop times are required. The number of floating point operations performed in parallel by these jobs varies from 4 to $2n^2 - 2$. Likewise for each of the n - 1 back substitution jobs 2 flop times are required. The number of floating point operations which can be done in parallel for these jobs varies from 2 to 2n - 2. The total time required for Gaussian elimination on this data flow machine is thus 5n - 4 flop times. Note however, that up to n divisions, or $2n^2 - 2$ multiplications or $2n^2 - 2$ subtractions must be performed in parallel. Thus, although the performance of such a data flow machine would be very high, its resource utilization is extremely poor, namely

$$\frac{\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n}{(4n^2+n-4)(5n-4)} .$$

Thus for a large n the resource utilization is approximately $\frac{2}{3} / 20$, i.e. $3\frac{1}{3}\%$. Thus it would be extremely expensive to construct a data flow machine which would exploit the full parallelism of the Gaussian elimination program.

Intermediate between the serial computer which is unattractive in performance and the maximally parallel data flow machine which is unattractive in cost are many practical realizations including instruction pipelining, specialized array or vector oriented pipelined floating point units embedded in a conventional architecture, as well as vector oriented, array oriented, and multiple processor organizations. All of these machines should be intermediate in hardware cost, performance level, and resource utilization when compared with the extreme machines. Further evaluation is required to determine precisely which of these computer organizations will come closest to the performance of the data flow machine with a cost more like that of the serial machine. The correct choice of machine will surely depend on the range of the expected values for n, the number of simultaneous linear equations to be solved. To be efficient, a preferred architecture must adapt efficiently to the variable vector size and the irregular degree of parallelism caused by the nature of the jobs and their precedence relationships in the Gaussian elimination problem. These requirements would tend to give preference to vector or pipelined architectures and multiprocessors over the relatively more rigid array machines. Finally, the memory hierarchy organization must be carefully constructed to preserve the high resource utilization expected of the selected architecture. Paging traffic could easily be of order n^3 for this problem. Intuitively, one could explain this amount of page traffic by considering that the data of the problem would contain of the order of n^2 pages each revisited on the order of n times.

3.9 Gaussian Elimination Experiments on the AMP-1 System

Three versions of a basic Gaussian elimination program were implemented for the AMP-1 multiprocessor system. The first of these is GAUSB which conforms to the description previously given. The second is GAUSY which is similar to GAUSB except that many precedence flags were used to reduce semaphore lockout. In the GAUSB version, all critical sections used a single semaphore. Thus no job could proceed in any critical section of code while any other job was in any other critical section of code. The GAUSY version used approximately 200 separate flags associated one-to-one with all possible critical sections. GAUSY is used to see if there is any significant performance degradation in GAUSB caused by access congestion at the single semaphore. Finally, GAUSZ is similar to GAUSY except that no normalize jobs are present. This change was made since the normalize jobs as shown in Figure 23 severely restrict the amount of parallelism in the Gaussian elimination program. The row reduction and back substitution jobs are then recoded to permit working with an A matrix diagonal that does not contain 1's. Elimination of the normalize jobs saves $\frac{1}{2}n^2 + \frac{1}{2}n$ divisions. Each row reduction job then contains 1 division and each back substitution job contains 1 division. A final division is required to compute the value of the last variable. Thus there is no change in the total number of divisions required by the Gaussian elimination program. However, there is now a much less restrictive job precedence relationship which results in more parallelism which can potentially be exploited in a multiprocessor. It must be remembered that GAUSB and GAUSZ would have the same execution time on a system with a single serial processor. This type of change reflects the kind of considerations which become important for multiple processor systems.

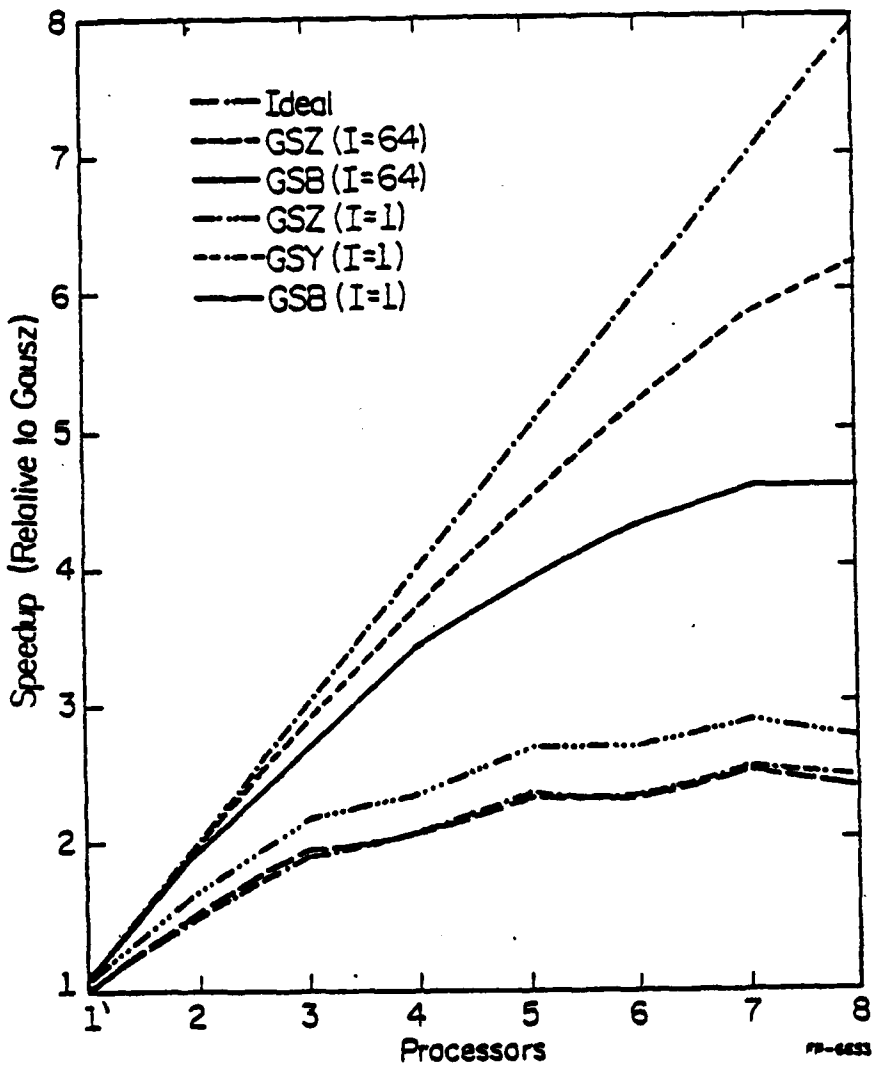


Figure 25. Gaussian elimination performance.

The data graphed in Figure 25 shows the differences in performance level between these three versions of the Gaussian elimination program for maximum and minimum degrees of interleaving as a function of the number of processors. It is apparent that little semaphore congestion exists in the GAUSB program since GAUSY has virtually identical performance to GAUSB. GAUSZ, however, is slightly higher in performance with no interleaving of the memory and is clearly superior when a sufficient degree of memory interleaving is present, particularly as the number of processors grows to a point where the additional parallelism of the GAUSZ program can be exploited. Figure 26 shows in detail the performance of GAUSZ for various degrees of interleaving as a function of the number of processors. Despite the job precedence restrictions which cause processor wait time, the critical sections of code which can be executed by only 1 processor at a time, and the memory access contention in the shared memory, a performance speedup of over 6 exists for 8 processors relative to the performance of 1 processor when the memory is fully interleaved, I = 64.

Specific data was collected to determine the extent of performance degradation due to memory access conflict. These data were compared against performance levels predicted by the memory access conflict model used previously for matrix multiplication. The GAUSZ program references memory slightly more often than the matrix multiply program. Modeled and measured probabilities of acceptance for an access request to memory are shown in Figure 27. The model conforms fairly closely to the measured levels, but not as closely as for matrix multiplication. The model tends to be somewhat lower than measured values for probability of acceptance indicating that memory contention is actual slightly less than the model predicts. This is at least partly due to the fact that the memory request rate actually decreases slightly as the number of processors increases, $Y = .7961$ for 7 processors. This decrease is due to the effect of specialized program codes executed while

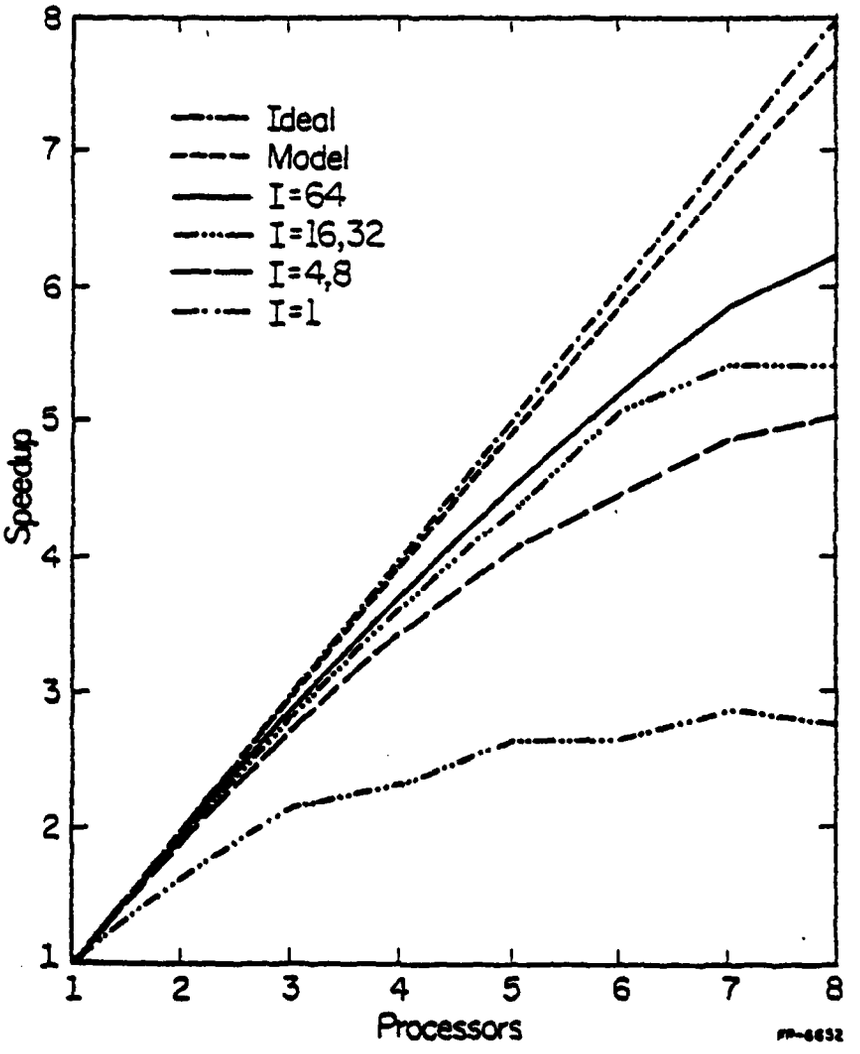


Figure 26. GAUSZ performance.

$\dagger$ = .8316 for GAUSZ (vs. .7917 for MXMC)

P	MODEL P_A	MEASURED P_A
1	1	1
2	.9936	.9950
3	.9873	.9897
4	.9809	.9844
5	.9746	.9823
6	.9682	.9827
7	.9619	.9769
8	.9555	-

Figure 27. Memory access conflict model for GAUSZ.

a processor is waiting to enter a critical section or waiting for job precedence to become satisfied. The performance degradation from ideal speedup caused by memory access contention is shown in the curve labeled Model in Figure 26. Thus one may conclude that the performance degradation of the GAUSZ program for fully interleaved memory is not primarily due to memory access conflict. Further research is required to accurately model and measure performance degradations due to the execution of critical sections which can be executed only by one processor at a time and those due to waiting for a prior job to be completed. This research is somewhat complicated by the fact that while a processor waits, it is actually executing code in a tight loop and making memory access requests as well while in this loop.

3.10 Conclusions and Recommendations

A preliminary characterization of finite element methods has been completed by measuring performance of actual state-of-the-art code for the finite element method on conventional medium scale and large scale computers. These results indicate that numerical integration, and specifically matrix multiplication performed in the numerical integration routines, and solving of linear equations are most significant in determining performance. As problems become larger and nonlinear, the proportion of time spent executing these two kernels of code becomes increasingly dominant. Paging traffic is also severe for these larger problems.

Known algorithms and studies on existing computers have been collected for matrix multiplication and solving of linear equations. The studies for matrix multiplication indicate that the side effects of certain vector and array oriented computers are severe and minor modifications of programs which would make no difference for a serial processor with matrices resident in

primary memory, make a great deal of difference in the performance of more highly concurrent architectures and paged systems. For the solution of linear equations, the Choleski and LU decomposition type algorithms are preferred for banded, symmetric matrices in linear equation solving. To be truly effective, these algorithms must be further tailored to the particular computer architecture being considered and must be amenable to dealing with large problems by accessing the matrices involved in blocks, so as to reduce the amount of paging required. Iteration and retriangulation effects for nonlinear equations must be considered as well.

Pilot multiprocessor experiments show that a high degree of parallelism is possible in these kernels of code. Appropriately structured shared memory can achieve low access conflict. Effective access conflict models exist. However, further research is required to extend the memory access conflict model to a general shared resource model which can effectively deal with memory access requests as well as shared function units, critical sections of code, and job precedence wait. This research has been initiated.

Also initiated is research oriented toward modeling alternative structures for high performance multi-access cache memories within a general memory hierarchy model. Such a model is essential for discovering effective organizations for the memory hierarchies of multiprocessor systems as well as for a single processor systems with high performance dedicated function units having direct memory access capability.

A general approach to characterizing alternative architectures and alternative algorithms is in the formative stages. Such a model will allow a single characterization for each algorithm regardless of the architecture on which it is to be run. This characterization will include direct measures of the types of parallelism inherent in the algorithms which can be exploited by

appropriate architectures. Such a model is extremely useful identifying the most effective algorithm-architecture pairs evaluation in detail.

Appropriate machine primitives are being identified for the matrix computations. These will include embedded vector operati address generation constructs, etc. Primitives to be develop compatible with memory overlays, sparse matrix computations precedence relationships. Once such primitives are identified, t instrumental in determining the effective instruction sets fo tailored to matrix computation as well as appropriate function uni computers.

The research described above can be completed within a thre frame with a funding level sufficient to support 3 faculty with assistants including funds for computer time and the development subsystems. A detailed budget along these lines would amount to a $150,000 per year for the 3 years.

The research emphasis for years 1 and 2 will be to complete and experimental studies for alternative architectures and algorit range of concurrent processing architectures, memory organiz function units will be considered. The cost effectivenes architectures and algorithms will be identified with respec application to the finite element method. Primary considerat evaluation will be given to large nonlinear and dynamic probl matrix approaches leading to low paging rates, and exploitable par efficient computation. Year 2 will also include further measure complex finite element problems with conventional archit state-of-the-art code. These results will provide a base l

evaluation of more effective architectures and algorithms.

In year 3 the emphasis will be on deriving recommended preferred architectures and the algorithms appropriate for them considering existing large scale computers, medium scale computers with enhanced memory and function capability, and highly concurrent processor and multiprocessor systems.

SECTION IV

EVALUATION OF SIGNAL PROCESSING ARCHITECTURES

4.1 Simulation Studies

In order to accurately select cost effective candidate architectures for signal processing applications, a number of simulation studies have been performed. Studies like these are an effective approach toward gaining a better understanding of computer system performance. Two simulators have been implemented; the first describes a shared resource multiprocessor, while the second describes a high speed vector processor. Each of these architectures uses parallel processing techniques to enhance the computational rate.

4.1.1 A Simulator for a Shared-Resource Multiprocessor

We have completed the design and construction of a simulator for a shared-resource multiprocessor (SRM). An SRM is logically similar to a tightly-coupled multiprocessor and contains multiple virtual processors that can simultaneously execute multiple, independent instruction streams (programs). These programs may, however, interact via explicit synchronization instructions. In actual fact, there is only one physical processor that is organized in a manner similar to a high performance uniprocessor such as a System/360 Model 91, i.e., it is overlapped and pipelined. Each virtual processor has, dedicated to it, a set of registers which are known as a skeleton processor. The skeleton processor holds the state of the corresponding virtual processor. The rest of the resources, including the instruction pipeline, the functional units, the buses and memory are shared by the virtual processors in a time-multiplexed fashion.

At no point in time does any one stream have more than a single instruction in any stage of execution. Thus, the problems associated with instruction lookahead, such as guaranteeing logical independence between concurrently executing instructions from the same stream, are eliminated. This contributes to simplicity in the hardware. On the other hand, it is possible to have multiple instructions from distinct streams (independent by definition) being executed simultaneously, thereby achieving concurrency. The SRM organization could potentially be very attractive from a cost-performance viewpoint for signal processing purposes. It is with this in mind that we initiated work on the simulator.

The simulator has been written in a high-level simulation language called SIMULA. SIMULA is based on ALGOL 60 but has been enhanced to facilitate simulation. The two most important features involve the addition of a co-routine capability and a limited language extension capability. The former supports the simulation of simultaneously existent objects in a natural way and the latter provides the ability to construct a simulation environment that is well matched to the application at hand. In SIMULA, both features are provided via the CLASS construct. We have found it to be extremely useful in developing our simulator.

One of the initial problems that we encountered in the design of the simulator arises from the fact that the system being simulated is essentially a parallel structure whereas programming languages are generally sequential. The existence of co-routines in SIMULA provides a form of parallelism that is termed quasi-parallelism. This reduces, but does not eliminate the problem of describing parallel structures and computations. We, therefore, developed a rather general programming construct, a graph which consists of a set of nodes with arbitrary precedence relationships between them. A node (currently) contains conventional sequential code within it. This code will be executed

only when the specified precedence conditions have been satisfied, viz., all predecessor nodes in the graph have completed execution. This graph construct has been incorporated into the simulator using the language extension feature of SIMULA. Although this construct is not the last word on this issue, it has greatly facilitated the construction of the simulator by providing the illusion of a more parallel language.

A second problem, which points out a serious deficiency in SIMULA for our purposes, is the lack of a rich variety of data types and operations upon them. SIMULA has the data types integer decimal, floating point decimal, boolean, etc., but not integer binary, integer hex, etc. This leads to significant problems in representing the contents of memory. If represented as a bit string (the most natural and desirable representation), arithmetic is difficult and must be simulated in detail in a bit-by-bit manner. If this is not done, instruction interpretation, field extraction and bit string manipulation are complicated. There is no direct solution to this problem in SIMULA. We plan to solve it by interfacing machine language subroutines to the simulator to support bit string manipulation.

We have developed a novel simulator organization which permits for great flexibility in comparing different computer structures with the same instruction set architecture, or different instruction set architectures using the same hardware organization. This has been achieved by implementing the simulator as two almost independent parts; one part corresponds to the instruction set interpreter and all information regarding the nature of the instruction set is localized here; the second part is concerned with the organization of the hardware and encapsulates all the details of the structure of the machine. We have been able to define a partition such that there is a very limited amount of interaction between the two partitions. As a consequence, it is easy, for instance, to replace the interpreter part by code

for another instruction set architecture, thereby obtaining the simulator for a machine with the same organization but a different instruction set.

For purposes of this investigation, we have developed a simple, register oriented architecture with powerful (PDP-11-like) addressing modes. Data and instructions are 32 bits wide each. The simulator for a shared resource multiprocessor with such an instruction set has been designed, implemented and debugged. It is highly parameterized to allow a number of variations to be studied. Included in the set of design parameters that we wish to examine are the number of instruction streams, the number of memory modules, the number of buses between the processor and the memory, the speed of the pipelined multiply unit, the memory cycle time and the number of slots per buffer. In addition, we intend to study various priority schemes to be used in selecting from a number of contending requests. The workload used will be a matrix multiplication algorithm since computations of this type are common in signal processing applications. As a result of these studies, we shall have gained valuable insight into the design and performance of an architecture that we feel is a good cost performance candidate for signal processing.

4.1.2 A Simulator for a Vector Processor

A simulator for an existing high speed vector processor was constructed in order to evaluate the effectiveness of existing vector processing methodologies. The architectures simulated consist of a class of computers closely patterned after the highly successful CRAY-1 processor. However, the simulation was constructed so as to allow the alteration of architectural parameters such as: the number of vector registers, the vector register length, the memory cycle time, the number of memory banks, the number of vector functional units and the instruction buffer size. By varying these

parameters, we can explore the performance of a class of processors related to the CRAY-1 for the execution of selected benchmarks from the application area. The actual simulation program was constructed in FORTRAN for lack of a better simulation tool. Rather than explicitly simulating actual logic, the simulator models reservations placed on the use of functional units and memory banks and tests instruction issue conditions to determine the readiness of the next instruction for execution. While the simulator accurately predicts the performance of memory resident CRAY assembly language programs, no attempt was made to simulate I/O. The accurate simulation of I/O was considered too difficult, especially in a serial programming language such as FORTRAN.

A matrix multiplication program was selected as a simple benchmark program for the investigation of this class of vector processors. It should be noted that this is a highly computation intensive program stressing demands on the vector floating point functional units of the CRAY-1. The benchmark program had to be reprogrammed when certain of the architectural parameters were varied in order to exploit added capability. For example, if the number of vector registers is increased, the benchmark program has to be reprogrammed to exploit this additional hardware. The use of additional vector registers may lead to higher performance since a vector fetch operation could be more efficiently overlapped with vector multiply operations. The first simulation experiment performed measured the the effects of the number of vector registers on benchmark performance. The standard CRAY-1 processor has 8 vector registers each 64 elements long. The innermost loop from the benchmark program was simulated on machines having 4,8, and 16 vector registers to estimate performance. The table below illustrates the results of this study.

Number of vect. reg's.	MFLOPS	%Change
4	135.33	-8.7
8	148.23	—
16	150.36	+1.4

Note that for this benchmark, the execution rate in million floating point operations per second is rather insensitive to the addition of new vector registers. This is partially due to the fact that the increased length innermost loop program for 16 registers will no longer fit within the instruction buffer and requires the instruction buffer to be repeatedly reloaded increasing memory traffic.

The second experiment explored the effects of vector length on processor performance. As the vector length is increased, the vector startup cost is averaged over a larger number of elemental operations leading to more efficient operation. The results of this experiment are shown below.

Vector length	MFLOPS	%Change
8	78.77	-38.8
16	105.13	-18.3
32	119.77	-7.0
64	128.72	—
128	133.73	+3.9

This experiment was performed using the complete matrix multiply program instead of the innermost loop used above. Thus, the simulated performance is somewhat lower than that shown in the experiment on the number of vector registers described above. From these studies, it became clear that this computationally intensive benchmark is largely limited by functional unit performance, and memory bandwidth.

The following experiment was completed to measure the dependence of performance on vector functional unit capability. The simulation was constructed to allow the vector functional units to operate on more than one element at a time. Thus, while the startup time for a vector instruction is constant, the execution rate of elemental operations is proportional to the number of parallel operations performed at a time (NPAROP).

NPAROP	MFLOPS	%Change
1	128.72	—
2	162.80	+26.5
4	189.28	+47.0
8	203.98	+58.5
16	206.45	+60.4
32	206.45	+60.4

Note that the performance of this computationally intensive benchmark program is highly sensitive to functional unit parallelism. However, when 16 or more floating point operations are performed simultaneously, the performance increases level off due to memory bandwidth saturation.

4.2 Architectural Issues for Fast Fourier Transform Processing

This project was the result of exploration of methods of Fast Fourier Transform (FFT) implementation and of machines that had been conceived to perform these operations. It is useful to consider FFT operations, because many signal processing algorithms and manipulations are FFT-like. These include the Discrete Fourier Transform (DFT), the inverse DFT, convolution, and correlation.

The basic form of the Cooley-Tukey algorithm resulted in savings of several orders of magnitude in computation of the FFT for moderate and long sequence lengths. Many methods of honing the time to compute FFT's have been proposed. Most of these methods capitalize on bottlenecks in hardware such as long multiply time with respect to add time, or on special cases related to applications, such as all real input data.

The methods for increasing computational speed through specialized signal processing hardware can result in a performance gain of perhaps two orders of magnitude over general purpose computers. This is as significant as the savings realized from the FFT algorithm, and should receive careful attention.

Implementation of special FFT processing hardware results in not only a computational savings, but often an economic savings as well. Many of the early considerations of hardware implementation have been discussed by Bergland [31]. One example of design implementation is given by Pomerleau, et al [32], for the realization of an FFT processor, based on real time, real valued input sequences, and an attempt to maximize the precision of the result.

The heart of all FFT signal processors is some form of FFT unit, or specialized ALU to efficiently perform the sums-of-products operations inherent in FFT computation. Parallelism and pipelining can be introduced at all levels of computation to enhance performance. The "butterfly", the basic primitive of Radix-2 FFT's, can be highly parallel-pipelined. To produce each of the $\frac{N}{2}Log_2N$ butterflies sequentially at a very fast rate. Also, if many "butterfly units" are replicated, as many as N/2 butterflies can be executed in parallel. By combining many parallel units that are each parallel-pipelined, the maximum speed can be achieved. Economic constraints, however, will limit this maximally parallel, pipelined structure to those cases where application dictates the absolute necessity of handling a large amount of data very quickly.

Today's technology allows a single butterfly to be computed in less than 100 nanoseconds. With this speed available, the general trend is to provide a single, very fast butterfly unit, and sequentially compute each butterfly. This speed is sufficient for many real-time applications.

Since the late 1960's, many specialized signal processors have been constructed. Two basic categories exist: First, the dedicated processor that operates as a stand-alone processor, and second, the distributed system that takes the form of a specialized peripheral controlled by a host computer. The

dedicated processors tend to be more flexible in their operations, often providing for data processing other than FFT's such as windowing, buffering, smoothing, interpolating, and automatic gain control. The distributed processors take advantage of the widely varying speeds between the FFT computation and the handling of the data sequence. The "number crunching" is handled by a special high-speed butterfly unit, while the slower host initiates tasks, and performs operations that require decision-making capabilities.

Several of the representative machines have been compared by Allen [33], and numerous articles relating to this can be found in the collection [34]. Current work includes implementation of machines to take advantage of numerous specialized algorithms and machines that take advantage of ECL, LSI, VLSI, and the advanced state-of-the-art technology.

The need to compute a 2 dimensional discrete Fourier Transform (2D DFT) of a large array (say 1024 x 1024 or larger) arises in many different practical problems. Unfortunately, to take a 2D DFT, even using a FFT algorithm, requires a large amount of computer resources (i.e., memory and CPU time). This research has been concerned with how such a DFT can be calculated most efficiently.

There is very little that can be done to minimize memory as almost all of the memory typically used is required to store the data array. An insignificant amount of the required memory is needed to store the program itself. One possible tradeoff between memory and speed is whether or not to store a table of constants need in the FFT butterfly operations. Since the time penalty of calculating rather than storing the needed constants is so great, we assume the needed constants are stored.

The procedure for computing a 2D DFT of an N X N array is as follows. N-N point 1-dimensional DFT's are calculated along rows or columns. The array is then transposed and the process is repeated. Thus, the number of operations required for a straightforward DFT implementation will be proportional to N^3. If a FFT algorithm is used, the complexity is reduced to $N^2 \log_2 N$.

When working with large arrays, the entire array may not fit into main memory. In most cases it will be stored in row major or column major order on some sort of sequential access memory device such as a disk. In the process of calculating the DFT, each row or sequence of rows will be read from disk into main memory where the 1-dimensional DFT's will be calculated. The result will then be returned to the disk and the process will continue. The penalty of doing this will be relatively small since each 1-dimensional DFT represents a large amount of computation.

The problem arises when transposing the array. It is obvious that transposing an array stored on a disk in a straightforward manner would be very time consuming as the number of read and write operations would be approximately equal to the number of elements in the array.

On the other hand, methods such as the one proposed by Eklundh can reduce the required number of I/O operations significantly [35]. For instance, if an array contains 2^n X 2^n samples, the array can be transposed without reading in and writing out the array more than n times, assuming that at least 2 rows of the array fit in main memory at once. If a larger number of rows will fit into main memory at once, the number of times the array will have to be read and written can be reduced to as little as two.

Another problem is that of reducing paging faults to a minimum. Unfortunately, because of the sequence of operations, there is very little that can be done to reduce page faults. Ideally, all the pages that hold a given row should remain in main memory until the DFT on that row has been completed. Otherwise, an excessive number of disk swaps will be necessary.

Another related area investigated is the calculation of a 2D DFT where the input data is in polar rather than rectangular coordinates. This is a problem that often arises in synthetic aperture radar, tomography, and crystalography. The traditional approach has been to use some method of interpolating the data into rectangular coordinates and then calculate the transform in a conventional manner. This, however, is very time consuming and results in large numerical errors. Another procedure is to manipulate the transform into a finite integral which can be evaluated numerically. This can also be very time consuming. We have begun a research study to derive a discrete form of the polar Fourier Transform. The polar Fourier Transform pair is

$$F(\rho,\phi) = \int_0^{\infty}\int_0^{2\pi} f(r,\theta)\exp[-j2\pi r\rho\cos(\theta-\phi)]\,r\,d\theta\,dr$$

and

$$f(r,\theta) = \int_0^{\infty}\int_0^{2\pi} F(\rho,\phi)\exp[j2\pi r\rho\cos(\theta-\phi)]\,\rho\,d\phi\,d\rho\,.$$

The presence of the cosine term in the experimentation makes the polar Fourier Transform much more difficult to evaluate than the rectangular form. In the process of deriving a DFT from a continuous form, it is necessary to know several transform pairs from the continuous transform. No such tabulation for polar Fourier Transforms is now known to us. We are hopeful that we can develop this tabulation, that the polar Fourier Transform will lend itself to a discrete form and that a "fast" implementation will be possible.

SECTION V

AVIONICS PROCESSOR ARCHITECTURES EVALUATION

In real time computer applications that require the concurrent handling of many tasks, computational efficiency is very important. Specifically, in the avionics environment, where navigation, system monitoring and weapons delivery are included among the ongoing tasks of the avionics computer, efficiency of computation is the foremost requirement for handling the voluminous data entering the computer. Computational efficiency results from a combination of the system architecture and the software used to control the many processes. A given task domain has inherent processes which can give rise design possibilities for tuning both hardware and software in order to optimize the overall efficiency of a processing system. In particular, expected types of data flow, operations, and computation sequences that recur frequently can lead to the specification of data paths and instruction types, which if added to an existing architecture can considerably improve its throughput. In addition to this, compilers that are optimized with respect to the architectural features can greatly improve the processing efficiency.

We have begun an investigation of the processor efficiency of the Air Force AYK/15A computer with respect to avionics processing requirements. The intention is to find areas which can be improved and to investigate the consequences of proposed improvements. This investigation is being done in two steps. The first step is to compare the architectural features of the AYK/15A with those of the Raytheon fault tolerant space borne computer (FTSC) and with those of the Delco Magic 362F. Simulators which run on the DEC System-10 have been constructed for these machines as tools on which benchmarks of representative avionics processes can be run. The second step is to code the benchmarks on each machine and evaluate the runs with respect to instruction and address mode usage, memory reads and writes, storage

requirements, and register usage.

5.1 The Avionics Processors

5.1.1 The Air Force AYK/15A

The AYK/15A is an extension of the AYK/15 prototype Westinghouse in conjunction with AFAL. The AYK/15A has instruction set of the AYK/15 [36,37]. It has up to 65,536 core memory. In addition it has 16 user accessible 16-bit re which can be used as accumulators, stack pointers, index registers, and temporary storage. It has 207 implemented length 16 bits and 32 bits. It has the following address mode

Register	EA = Reg
Direct	EA = Address
Direct-Indexed	EA = Address + (Rx)
Indirect	EA = (Address)
Pre-Index Indirect	EA = (Address + (Rx))
Immediate Long	(EA)= Address
Immediate Short with Positive or Negative operand	(EA)= sign-extended 4-b
IC-Relative	EA = (IC) + Displaceme
Base Relative	EA = (BR) + Displaceme
Base Relative-Indexed	EA = (BR) + (Rx)
Special modes	

(Rx : R1-R15; BR : R4-R7; IC—Instruction Counter)

Subroutine Linkage

The AYK15A provides several ways for calling subroutines.

1. JS Ra, Label return by JC 15,0,Ra

The return address is stored in Ra and subroutine par be passed through STM and LM which stores and loa registers respectively.

LM n,0,Ra will bring in all parameters and

STM m,0,Ra will return the results.

2. SJS Ra,Label return by URS Ra

The return address is stored in the stack location pointed to by Ra. This saves one address word for the return instruction. Subroutine parameters may be passed by PSHM and POPM which pushes and pops multiple registers respectively onto the stack pointed by R15. PSHM and POPM can act on individual register too. Thus, it frees the registers to be used in subroutines.

Unusual but Useful Features

The CBL instruction tests if (Ra) is less than, in between, or greater than an integer-interval defined by (Addr) and (Addr+1). Base-Relative Addressing Mode allows single word instructions (e.g. ADD, SUB, AND, OR, etc.) for record structuring in high level programming languages with the base register pointing at the record. This decreases the program size and speeds up the execution as well. It is to be noted that this address mode is available for certain registers only.

General Comments

The AYK15A is a general purpose processor and it has very well designed instructions. Most of the short instructions (e.g. Immediate-Short, Base Relative, ADD LOAD) are the most commonly used instructions. Thus it results in less memory fetches and faster execution as well as smaller program size.

The instructions PSHM and POPM can free any number of consecutive registers for use in subroutine linking. There are 16 registers available for users and even though they are 16 bits long, they are sufficient for addressing all of memory.

The AYK15A lacks some rather important function instructions such as square-root, vector-manipulations which are common in aviation formulae.

Since the machine is 16-bits but floating point data are in 32-bits, this means that a floating point array indexing have to be doubled. This can be done just by a logical left shift. This may be the reason that AYK15A has no post-increment index address mode. The processor has the one-word instruction to add two to an indexing register and this may be the replacement the designer put up for the lack of post-increment index mode.

5.1.2 The Raytheon Fault Tolerant Spaceborne Computer

The Raytheon Fault Tolerant Spaceborne Computer (FTSC) has the generality of a conventional computer, with many added features for hardware and software fault detection and recovery. It has an instruction space of 128 instructions with 112 implemented [38]. Of these, 18 are operational in executive mode only. The word size is 32 bits and the arithmetic is two's complement.

The FTSC has eight 32-bit general purpose registers that are programmer accessible and eight 32-bit working registers. Four of the working registers have special assignments, namely, the memory data register, the memory address register, the status register, and the extension register. The status register contains the following information: bit 8 is the carry-out flag, bit 9 is the invalid arithmetic operation flag, bit 10 contains the overflow status, bit 11 contains the executive mode status, bit 12 is the interrupt disable, bits 13-15 contain the interrupt level if an interrupt is running, bits 16-31 are the program counter. The extension register is used with double word operations such as long shifts and floating point instructions to accomodate the least significant bits. The other four registers have no special designation and are used as scratch registers by the more complex

instructions.

Data Formats

Logical data are stored in one word and each bit is treated identically.

Integer data is stored in one word and has two's complement representation.

Single precision floating point data takes one word. Bits 0-23 are used for the two's complement normalized mantissa and bits 24 - 31 are used for the two's complement exponent. All floating point instructions expect normalized operands. Floating point zero is represented by all zeros in the mantissa and an 80 (hex) exponent.

Double-Precision Floating Point data takes two words. The high-order word has the exact format as the floating point data. The low-order word is a 32-bit continuation of the high-order mantissa. Double precision instructions expect normalized operands. Normalized double precision zero is a normalized floating point zero in the high-order word and all zeros in the low-order word.

Immediate numbers are treated as 16-bit integers. The sign bit occupies bit 16 and is extended by immediate mode instructions to the upper 16 bits of the word before the value is used in the computation. Immediate numbers may be used as logical or integer data.

Upper immediate numbers are treated as 16-bit two's complement integers. The value of the instruction address field is multiplied by 2^{16} before the value is used in the computation. Upper immediate data may be used in logical, integer, or floating point instructions.

The Instruction Set

Addressing modes:

AM	Name	Effect
0	Register	EA = Register
1	Immediate*	(EA) = Constant
1	Upper Immediate*	(EA) = Constant*2^{16}
2	Direct	EA = Address
3	Indirect	EA = (Address)
4	Indexed Postincrement	Reg = Reg + 1; EA = (Reg) + Address
5	Indexed Predecrement	EA = (Reg) + Address; Reg = Reg - 1
6	Indexed	EA = Address + (Index Reg)
7	Index Indirect	EA = (Address + (Index Reg))

*Addressing mode 1 is used only in the load type instructions (opcode 00-3F hex).

Instruction Format:

The FTSC machine has only one instruction format. Each instruction occupies exactly one word. Bits 0-6 contain the op-code. Bits 7-9 and bits 10-12 contain the RB and RA fields, respectively. The register fields, RB and RA, each specify one of the eight general purpose registers. For certain instructions, either one, or both, of these fields may be unused. Bits 13-15 contain the addressing mode. Addressing mode 1 is not used for store type instructions. Thus, when decoding an instruction, if the addressing mode is 1, the op-code is interpreted as if bit 0 were zero, thus forcing a load type instruction. Then when the effective address is being computed, bit 0 specifies whether the addressing mode is Immediate or Upper Immediate. Finally, bits 16-31 contain the address. Again, not all the instructions use the address field.

Useful unusual instructions:

SQUARE ROOT FLOATING
VECTOR ADDITION FLOATING
VECTOR SUBTRACTION FLOATING
VECTOR MULTIPLY FLOATING
VECTOR INNER PRODUCT FLOATING
VECTOR-SCALAR MULTIPLY FLOATING

Subroutine Linkage

The only way to save the program counter is by the jump to subroutine instruction. The effect of this instruction is to store the PC in the specified register and jump to the specified effective address. Once arriving at the subroutine the return address could be stored, stacked, or left in the register. The subroutine could then return through any of the several jump instructions used in any of seven addressing modes. It should be noted that there is no instruction to explicitly stack the PC or dynamic link when executing a jump to a subroutine. This scheme demands that the programmer take care in keeping track of his calling points if he wants his program to return properly from subroutines.

Interrupts

The priority interrupt network recognizes 10 levels of interrupts. In order of increasing priority these interrupts are: Direct memory access no. 2 (end-of-block), Direct Memory Access no. 1 (end-of-block), Serial Interface Unit (end-of-block), Direct Memory Access no. 2 (general), Direct Memory Access no. 1 (general), Serial Interface Unit (general), Real Time Interrupt, Arithmetic Error, Illegal Operation Code, and Fault.

The lower priority interrupts are honored during the instruction fetch cycle. The Illegal Operation Code interrupt and the Fault interrupt are honored at the end of every micro-code instruction. Interrupts 1-8 (the lower priority interrupts) are individually maskable under program control. Also it is possible to enable or disable the lower priority interrupts under program control. An interrupt is serviced if, and only if, it is not disabled or masked. When an interrupt is serviced, the program status register is stored in a preassigned memory location corresponding to the interrupt. Then the machine vectors to a location in memory according to the value stored at another preassigned memory location. Also the request flip-flop for the level serviced is reset and the "in-process" flip-flop for the same level is set.

While an interrupt is being serviced, all 8-lower priority interrupts are disabled. Only illegal operation codes and faults can interrupt. The in-process flip-flop can be reset by the return from interrupt instruction.

General Comments

The main strength of the FTSC is its ability to detect and correct errors, both in hardware and software. This is an advantage to the avionics programmer, since much effort is devoted to these problems in an avionics computer. Another strength lies in the speed gained from the inclusion of hardware vector arithmetic instructions. The FTSC was designed to calculate three dimensional vector algorithms. Another strength is the wide variety and uniformity of addressing modes; each load instruction has nine addressing modes, and each store instruction has seven.

The main weakness of the FTSC is the way it implements Upper Immediate addressing. Since the store instructions can never use immediate data, addressing mode 1 implies a load type instruction. The FTSC used this fact in

coding Upper Immediate by placing a 1 in the most significant bit of the opcode. Hence, hardware must recognize the address mode 1 concurrently with the 1 in the first opcode bit in order to distinguish the instruction from a store instruction, which also has a 1 in the first opcode bit.

5.1.3 The Delco Magic 362F

The Delco M362F is a modular, flexible, high performance digital computer [39,40]. It is a microprogrammed, high speed, general purpose, parallel computer with a 16 bit basic word length. It employs 16 and 32 bit instruction words and 8, 16, 32 and 64 bit data words. Multiple memory words are used for extended instructions and floating point and double precision data words. Two 8 bit bytes are stored in each memory location. Mainframe memory options include core and semiconductor. The arithmetic operations are binary, with negative numbers in the two's complement form. The processor is mechanized with standard and medium scale integration (MSI) and TTL integrated circuits. The MSI includes a 64 bit random access memory, a 2,048 bit read only memory, and a programmable arithmetic unit. The M362F addressing modes include direct (512 words), indirect, relative (index registers and instruction counter), and stack processing. The maximum memory size for the M362F is 65,536 words. The M362F processor has 16 user accessible registers that are used as accumulators, index registers, stack pointers and temporary storage.

Addressing modes

Direct	EA = Address
Indexed	EA = Address + (Index Reg)
Deferred	EA = (Address)
Index Deferred	EA = (Address + (Index Reg))
Post Indexing	EA = (Address) + (Post-Index Reg)
Index Deferred with Post Indexing	EA = (Address + (Index Reg)) + (Post-Index Reg)

Data Formats

FIXED POINT

Single precision	16 bit two's complement.
Double precision	32 bit two's complement.

FLOATING POINT

Single precision	32 bit (23 bit mantissa, 8 bit exponent)
Double precision	64 bit (48 bit mantissa, 8 bit exponent)

(Note: Both mantissa and exponent are also represented in two's complement.)

Subroutine Linkage

Subroutine linkage on the M362F is accomplished via a stack mechanism. To call a subroutine, the current Instruction Counter (IC) is pushed onto a stack located in memory via any register designated as the stack pointer. A transfer is then made to the specified routine. To return from the subroutine the IC is loaded form the stack area, which effects a transfer back to the calling routine. There are also instructions in the M362F repertoire which facilitate register stacking in memory, thus allowing registers to be used freely by the subroutine and then restored to their previous values before returning to the calling routine. It should be noted, however, that there is no instruction that pushes or pops multiple registers. Registers must either be pushed and popped individually (which requires tedious and inefficient coding) or stored and loaded en masse via the store and load multiple register instructions (which lacks some of the conveniences of a general recursive push-pop multiple facility).

The Instruction Set

The M362F instruction set is composed of 92 basic machine instructions, with 30 additional special purpose input/output and control instructions. There is direct addressing to 512 words, with relative, stack, indirect, and indirect/post-indexed addressing to 65,536 words. There are five instruction types. These include two 16 bit formats to operate on register and memory contents. One of these types is indexable, whereas the other is not. A third 16 bit instruction type is used to manipulate register data or to perform operations on register pairs. Finally, there are two 32 bit instruction types, both of which are indexable. The first is used to operate on memory and register contents and to process single bits in memory. The second is used to perform program transfers, compare immediate, load immediate, and to operate on register and memory contents. The M362F has some macro-instructions which are useful for avionics computations. These include:

PEX	polynomial expansion
PEXO	odd polynomial expansion
SSQ	sum of square
SQR	square root

Interrupt Processing

The occurrence of a specific M362F interrupt forces the processor to execute a unique transfer-to-subroutine instruction (TRSI). This method automatically transfers the program to the required interrupt servicing routine. The rules of interrupt operation are as follows:

1. An interrupt signal is not acknowledged until execution of the current instruction is completed. If that instruction is an IND or IPX instruction, the interrupt signal will not be acknowledged until execution of both that instruction and the next sequential instruction is completed.

2. Simultaneous interrupts are recognized in order of priority.

3. An interrupt routine may be interrupted by one of higher priority.

4. The responsibility of saving and restoring pertinent register data (except for the instruction counter contents) lies with the interrupt servicing routine.

5. Each interrupt servicing routine shall be terminated with a return-from-interrupt instruction (RFI).

6. At power-on, all interrupts except the power-off interrupt are disabled.

General Comments

The weaknesses of the M362F start in the complex layout of the opcodes. Many commonly used instructions are two words instead of one (a normal instruction size). A key example are the LRM and SRM instructions (load register from memory and load register to memory). These useful instructions need twice the size of less needed instructions like FSDP (double precision subtract).

Another problem is the lack of orthogonal design. The AND instruction ANDs the AREG to memory. The OR instruction will OR any register to memory. A programmer may, without thinking, assume that because the OR instruction will OR any register, the AND instruction will also. This means that the programmer must generally do an accumulator save, do a register transfer to accumulator, do the AND operation, transfer the result back to the register and then load the accumulator again. There are several instances of this inconvenient design feature.

The last major difficulty is when you want to do indexing or post indexing. Before any instruction wishing to index or post index you must precede that instruction with a IND or IPX. Other than just being annoying,

it can cause programmer errors if the program is modified. In haste the programmer may delete some assembler code and accidently leave in the IND or IPX. This would then effect the code immediately after the deletion and cause unknown behavior.

5.2 The Avionics Processor Simulators

In an attempt to make comparisons of the memory requirements, memory and register transfers, and speeds of execution for several benchmarks on the AYK/15A, the FTSC and the M362F, simulators were produced for these machines. A running version of the AYK/15A, which was generated from the ISP compiler developed at the Coordinated Science Laboratory, was obtained from AFAL. In addition, ISPS, which is a more powerful simulation construction facility, that allows easy gathering of statistical information about a simulator, was obtained from Carnegie-Mellon University. Along with this facility we obtained the ISPS version of the AYK/15A from CMU. ISPS descriptions were generated for the FTSC and the M362F. Debugging of these simulators and attempting to get support software to run with these simulators consumed the bulk of the effort for this part of the project.

The AFAL cross assembler was copied to the DEC-10 at CSL along with a loader for the AYK/15A and a lookup table for the M362F. Math packages were obtained for each of the machines. A FORTRAN version of the FTSC was obtained from SAMSO. It turned out to be unusable as it was written for a CYBER, and the conversion effort required to get it running on the DEC-10 was beyond our means. Instead we constructed a lookup table for the FTSC to be used with the the ALAP cross assembler.

A benchmark was written in FTSC assembly. This benchmark decomposes a square matrix into a lower and upper triangular matrix using Gaussian elimination. It is based on the algorithm suggested by CMU to exercise the floating point instructions and test the array addressing and nested iteration capabilities. Attempting to assemble the program brought to light several deficiencies in the ALAP cross assembler. ALAP is currently insensitive to word sizes larger then 16 bits. It allows the specification of larger word sizes, but attempts to either ignore this face, or to treat a 32 bit word as two consecutive 16 bit words. Conversations with personnel at AFAL and TRW at AFAL resulted in a new ALAP cross assembler being generated. However, this still did not properly generate values indicated by the DATA statement when the value was 2^{16} or larger. As a result, these values had to be hand coded into the simulated memory for the FTSC. Preliminary execution time, program size, and number of memory accesses were obtained.

This same benchmark was then coded for the AFAL simulator for the AYK/15A. Running the benchmark was inconclusive as the timings were not available for several of the instructions. However, some comparisons can be made between the FTSC and the AYK/15A. The FTSC used about the same number of words as the AYK/15A (78 vs. 80, respectively). However, since the FTSC is a 32 bit machine, it requires twice the number of bytes as the AYK. Examination of the FTSC machine code revealed that about 35% of the instructions had wasted lower bytes and were of no use to the program. Furthermore, the AYK accessed memory about 2/3 as many times as the FTSC. This may be due to the fact that the FTSC has only eight user accessible registers as opposed to 16 for the AYK/15A.

There is still some effort needed to complete the evaluation of these machines. A working version of ALAP must be constructed to allow cross assembly to 32 bit machines. It would be useful if the cross assembler

capabilities were extended to allow conditional coding based on the testing of a value in the address fields, as for instance, the address mode field. This would allow proper coding of the upper immediate mode for the FTSC instruction set. When the support software for the simulators is functional then the benchmarking should be performed.

SECTION VI

A HARDWARE SYSTEM FOR ANALYZING IMAGE PROCESSING KE

In order to adequately investigate the performance of va for image processing, it is necessary to work with rea resolutions comparable to those used in real applications. development of the CCD (charge-coupled devices) industry, it the very near future, CCD imaging devices will all b conventional vidicon camera. The advantages of CCD cameras a

1. accuracy and stability in the positions of the pixel
2. insensitivity to surrounding electric and magnetic f
3. virtual nonexistence of image bloom and lag, and
4. extreme ruggedness due to all solid-state constructi

Since CCD cameras with large arrays of pixels have recently from General Electric and RCA, it was decided to obtain th for research. Because it was more suitable for computer TN-2500 CCD camera from GE was selected. The camera featu array of pixels with the video signal quantized to 256 leve rate of 30 frames per second. Two of these cameras were experimental algorithms for stereoscopic vision could be inv

Next, the problem of interfacing the cameras to PDP-11/40, was considered. The problem was considerably choice of camera; nevertheless, with the pair of camer kilobytes of data at a rate of 4.5 megawords per second, it output could not just be dumped into the computer's memory b both the capacity and the bandwidth of the memory. For t decided to place an image buffer between the cameras and further advantage of having an image buffer is that it ca

memory bank for several processors. This requires a rather special design for the image buffer, but the possibilities for improved throughput make it well worth the effort.

An attempt was made to find a commercially available buffer memory meeting the above requirements, namely, a 4.5 megaword/second input bandwidth and an output port which could be easily multiplexed among several PDP-11's. However, nothing could be found satisfying both requirements, so it was decided to use a general purpose memory system to which we would add our own input and output ports. Even at that, the rather high input bandwidth narrowed the search for such a memory system down to just one, the Intel IN-7000 series. Each IN-7000 memory board contains 16 kilowords of static random access memory with a 250 nanosecond cycle time. Four boards would be required to store one pair of images from the stereo camera system; however, to further improve the system performance, eight boards were purchased so that a new image could be loaded into one part of the image buffer while the previous image was still being processed. In essence, two image buffers would be constructed and connected to common input and output ports via two electronic switches as shown in Figure 28.

The next consideration was how to allow several processors to simultaneously access the image buffer. Immediate concern was given only to the problem of access by three PDP-11's (one model 11/40 and two model 11/04's) which are presently available at CSL. However, the design was required to be flexible in the actual number that could be connected. It was further assumed that most requests for accessing the image buffer would come in short high speed bursts, i.e., a processor would fetch, say, 72 pixel values from the buffer at a time, then not request anything at all for a couple hundred machine cycles. With such an operating environment, a simple polling scheme would be suitable for arbitrating the memory requests from the

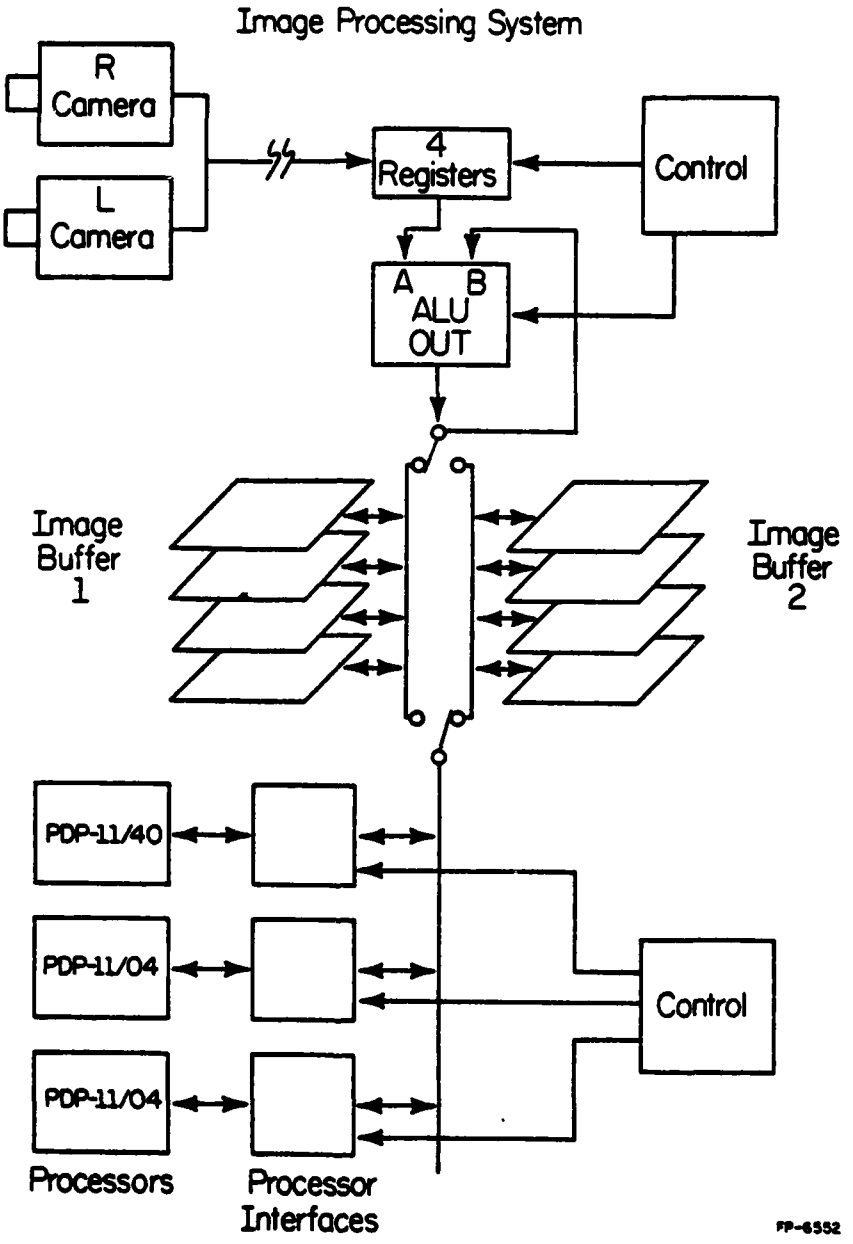


Figure 28. Interconnection of cameras, image buffer and processors.

several processors. Each processor, in its turn, would be polled to see if it required access to the image buffer, and if so, it would be granted a 250 nanosecond memory cycle. With individual requests from a single computer coming at a maximum rate of about one per microsecond, four computers could be simultaneously reading from the buffer at their maximum rates.

The input side of the image buffer consists of four 16-bit registers to hold four successive cells of pictorial data before being loaded into the memory array. Each register corresponds to one of the four memory boards used to store an image, and when all four registers are filled, their contents are transfered en masse into the memory array. It was later decided to route the second stage of this transfer through an arithmetic logic unit (ALU) so as to enable integration and differencing between successive image frames.

The purpose of the ALU is to modify the data coming from the cameras as it is being loaded into the buffer. The ALU operates on two operands labelled A and B. Input A is the data coming from the cameras, and input B is the data from the image buffer that is being overwritten. Of special interest was the operation in which the image coming from the cameras would be subtracted from the image already stored in the image buffer. It was desired that this be performed on either the left image or the right image or on both of them simultaneously. Thus the ALU would have to consist of two independent halves, each operating on a pair of 8-bit operands. Another desirable function was to add a series of successive frames together in order to obtain more bits of grayscale. For this, the two halves of the ALU would have to operate as a single 16-bit accumulator with the input from the left hand camera being ignored.

The primary consideration in the design of the ALU was speed. Whatever was going to be done had to take place in about 110 nanoseconds per pixel for the following reason. A new word of data comes from the cameras at the rate of one every 220 nanoseconds, thus filling the four registers at the input of the image buffer once every 880 nanoseconds. During those 880 nanoseconds, four pixels would have to be read from the image buffer, modified by the new data in the four registers, and stored back in the buffer. The only way this can be accomplished is to use two additional features of the Intel memory system:

1. the read-modify-write cycle which takes 60 nanoseconds less than a separate read cycle and write cycle, and
2. the ability to overlap the cycles of different memory boards on the same bus.

The manner in which the cycles overlap is shown in the following diagram:

```
BD 1   RRRRRRRRRRR MMMMMM WWWWWW ****
BD 2   RRRRRRRRRRR        MMMMMM WWWWWW ****
BD 3   RRRRRRRRRRR               MMMMMM WWWWWW ****
BD 4   RRRRRRRRRRR                      MMMMMM WWWWWW ****

       .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
TIME   0   100  200  300  400  500  600  700  800  900
```

where R represents the time it takes the data to reach the output of the memory board after initiation of the read-modify-write cycle; M represents the time this data is transmitted on the read-data bus; and W represents the time when the board samples the data on the write-data bus. The asterisks denote the remainder of the cycle to allow the write-data to completely sink into the memory chips. The inputs to the ALU become valid about 30 nanoseconds after the the beginning of the "modify" part of each cycle, but due to propagation delay, the output does not become valid until about 50 nanoseconds later. Finally, it takes about 20 nanoseconds to latch the output

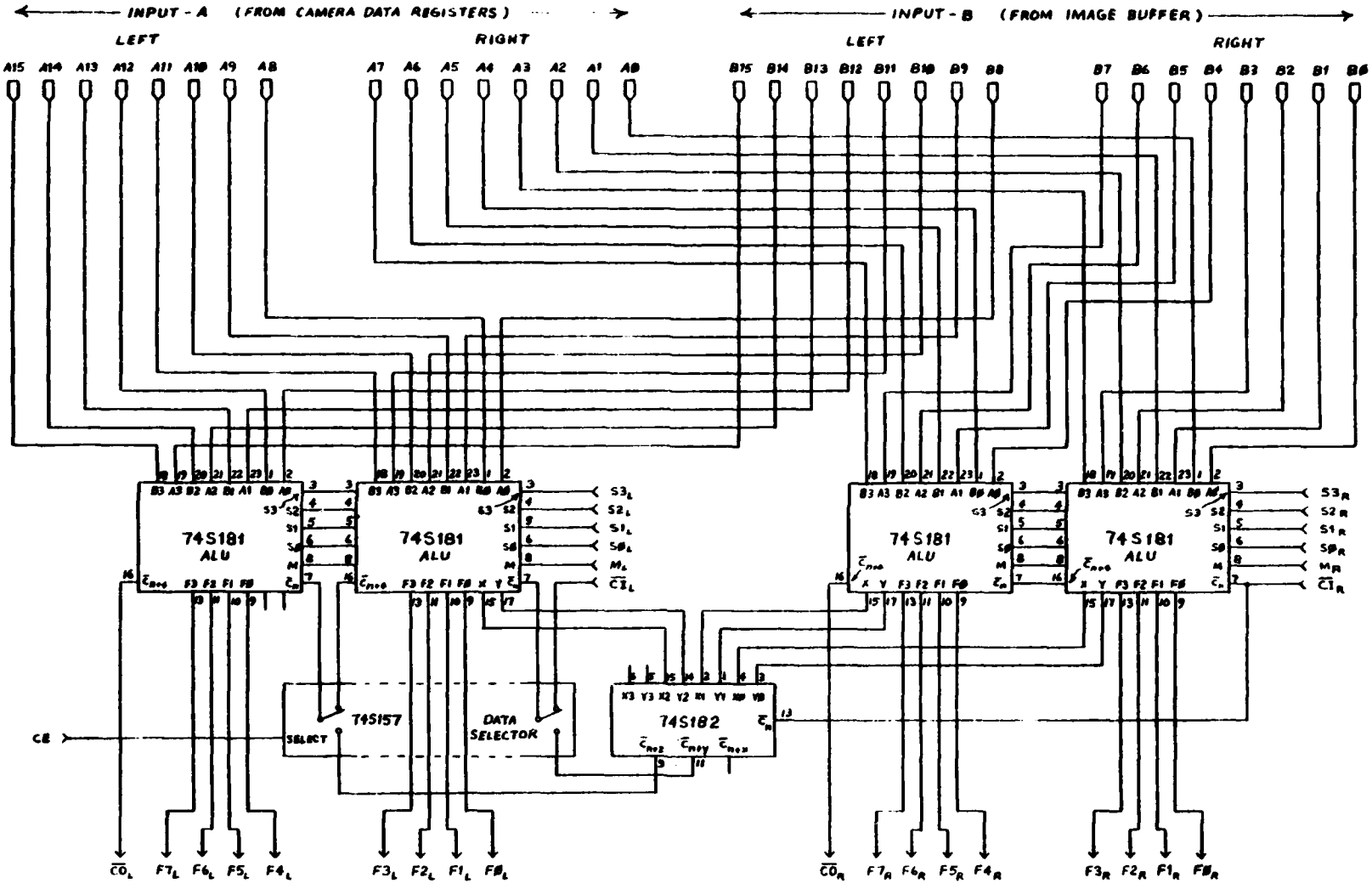


Figure 29. ALU circuit schematic.

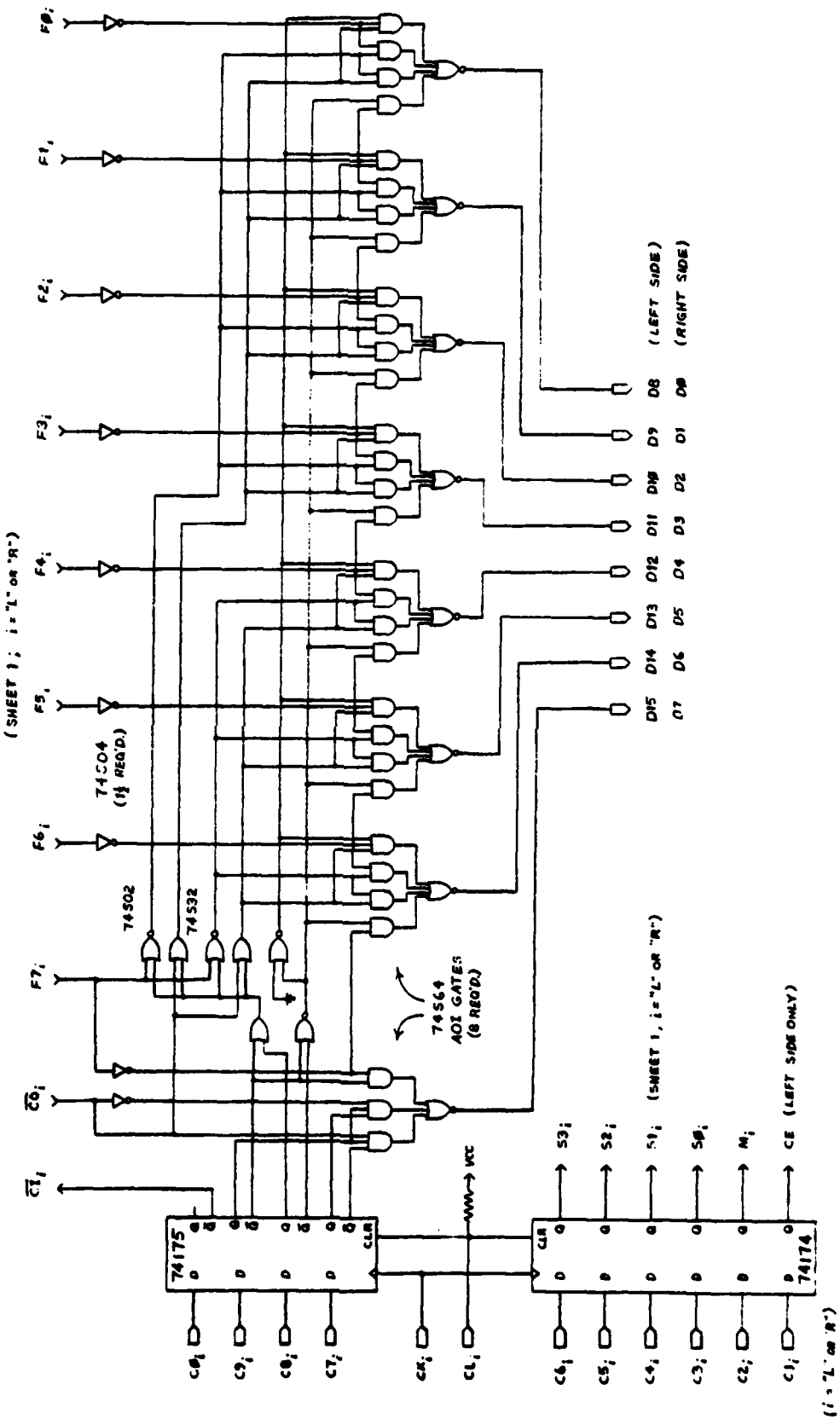


Figure 30. One half of the ALU section.

of the ALU to hold it steady for the subsequent "write" part of the cycle. Thus, the modified data to be written back into the memory board does not become available until 110 nanoseconds after the data first appeared at the beginning of the "modify" part of the cycle. No other board can use the read-data bus during this time, so the completion of each board's cycle is delayed 110 nanoseconds from the preceding board. For four boards, this just barely fits within the allotted 880 nanosecond period.

Hence, the basic ALU could have a total propagation delay of at most 60 nanoseconds. To achieve such performance, the design made exclusive use of Schottky TTL. The completed design, shown in Figures 29 and 30, has a calculated worst-case delay of 54 nanoseconds.

The ALU consists of two halves operating on 8-bit operands, though, upon command, they can also be connected to operate as a 16-bit ALU. Each half of the ALU is further divided into two stages, the first consisting of a pair of 4-bit ALU chips (the 74S181) which perform addition, subtraction, and Boolean logic, and the second consisting of discrete logic to compress the results from the previous stage into eight bits. Both halves of the first stage are shown together in Figure 29, while only one of the halves of the second stage (since both are identical) is shown in Figure 30.

A total of 20 bits are used to control the ALU, i.e., to specify what function the ALU is to perform. These are equally divided between the left and right halves of the unit, each half being controlled by ten bits labelled C0, C1, C2, ..., C9. The functions controlled by these bits are as follows:

C0 — the value of the carry input to the least significant bit of the ALU chips (when enabled by C1 & C2).

C1 — selects between C0 (above) and the carry output of the right half of the ALU for the carry input (only for the left half).

C2 — selects the mode of the ALU chips, arithmetic or Boolean

logic. (The Boolean mode effectively disables C0 & C1.)

C3-C6 selects one of 16 arithmetic functions or one of 16 Boolean logic operations, depending on the mode. Among the arithmetic functions are addition, subtraction, and shift left by one bit. Among the Boolean logic operations are the bitwise NOT, AND, OR, and EXCLUSIVE-OR.

C7 selects whether or not to invert the sign bit.

C8 selects whether to (a) clamp the 9-bit result from the ALU chips to fit within an 8-bit field, or (b) simply select the high-order eight bits of the 9-bit result.

C9 enables C7 & C8, otherwise all second-stage circuitry is bypassed.

The above descriptions should be self-explanatory with the exception of C8. Ordinarily, adding or subtracting two 8-bit operands yields a 9-bit result. However, there are only eight bits available for storing the result, so one bit has to go. If two 8-bit numbers were added, then it would normally be best to simply eliminate the least significant bit. This would also be a suitable alternative for subtraction, but when the difference is most likely to be small, as it will since two successive images are usually quite similar, then it would be better to retain the full resolution of the difference so long as it fits in an 8-bit field. This would include all results from -128 to +127. However, it is possible for the difference of two 8-bit operands to lie anywhere from -255 to +255, so something will have to be done with those differences below -128 or above +127. Changing these out-of-range values to -128 and +127, respectively, is known as clamping and is the other choice offered by control bit C8. If neither clamping nor selecting the high-order eight bits is desired, as would be the case if no modification to the camera data were desired, then bit C9 can be used to disable C8.

REFERENCES

1. McClellan J. H. and R. J. Purdy, "Applications of Digital Signal Processing to Radar," in Applications of Digital Signal Processing, A. V. Oppenheim (ed.), Prentice-Hall, Englewood Cliffs, N. J., 1978.

2. Swartzlander, E. E., "High Speed Micro signal Processor Study," Technical Report AFAL-TR-77-63, TRW Defense and Space Systems, Redondo Beach, CA, April 1977.

3. Hunt, B. R., "Digital Image Processing," in Applications of Digital Signal Processing, A. V. Oppenheim (ed.), Prentice-Hall, Englewood Cliffs, N. J., 1978.

4. Roberts, L. G., "Machine Perception of Three Dimensional Solids", Optical and Electro-optical Information Processing, J. Tippett et al., eds., MIT Press, Cambridge, Massachusetts, 1965.

5. Duda, R. and P. Hart, Pattern Classification and Scene Analysis, John Wiley & Sons, New York, 1973.

6. Burr, D., "On Computer Stereo Vision with Wire Frame Models", Ph.D. Thesis, Dept. of Electrical Engineering, University of Illinois, 1978, 117 pp.

7. Rosenfeld, A. R. and Kak, A. C., Digital Image Processing, Academic Press, New York, 1976.

8. Robinson, G. S., "Edge Detection by Compass Gradient Masks," Computer Graphics and Image Processing, Vol. 6, No. 5, October 1977, pp. 492-501.

9. Kirsch, R., "Computer Determination of the Constituent Structure of Biological Images, Computers and Biomedical Research, Vol. 4, No. 3, 1971, pp. 315-328.

10. Pratt, W., Digital Image Processing, John Wiley & Sons, New York, 1978.

11. Yakimovsky, Y., "Boundary and Object Detection in Real World Images", JACM, Vol. 23, No. 4, October 1976, pp. 599-618.

12. Hueckel, M. H., "An Operator which Locates Edges in Digitized Pictures," J. ACM, Vol. 18, No. 1, January 1971, pp. 113-125.

13. Hueckel, M. H., "A Local Visual Operator which Recognizes Edges and Lines," J. ACM, Vol. 20, No. 4, October 1973, pp. 634-647.

14. Castleman, K. R., Digital Image Processing, Prentice-Hall, Englewood Cliffs, N. J., 1979.

15. Nevatia, R., "Locating Object Boundaries in Textured Environments," IEEE Transactions on Computers, November 1976, pp. 1170-1175.

16. Ramer, U., "An Iterative Procedure for the Polygonal Approximation of Plane Curves," Computer Graphics and Image Processing, Vol. 1, No. 3, November 1972, pp. 244-256.

17. McKee, J. W. and J. K. Aggarwal, "Computer Recognition of Partial Views of Curved Objects," IEEE Trans. on Computers, Vol. C-26, No. 8, August 1977, pp. 790-800.

18. Ohlander, R. B., Analysis of Natural Scenes, Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, April 1975.

19. Haralick, R. E., "Statistical and Structural Approaches to Texture," Fourth IJCPR, November 1978, pp. 45-69.

20. Galloway, M., "Texture Analysis Using Gray Level Run Lengths," Computer Graphics and Image Processing, Vol. 4, No. 2, June 1975, pp. 172-199.

21. Nevatia, R., "Depth Measurement by Motion Stereo," Computer Graphics and Image Processing, Vol. 5, No. 2, 1976, pp. 203-214.

22. Hall, D. J., R. M. Endlich, D. E. Wolf and A. E. Brain, "Objective Methods For Registering Landmarks and Determining Cloud Motions from Satellite Data," IEEE Transactions on Computers, July 1972, pp. 768-776.

23. Claire, E. J., "Bandwidth Reduction in Image Transmission," Proc. IEEE 1972 International Communications Conference, June 1972, pp. 39-8 to 39-15.

24. Brown, R. D., "A Recursive Algorithm for Sequency-Ordered Fast Walsh Transforms," IEEE Trans. on Computers, Vol. C-26, No. 8, August 1977, pp. 819-822.

25. "LSI Electronically Programmable Arrays (Configurable Polynomial Arrays)," Technical Report AFAL-TR-76-228, RCA and Adaptronics, June 1977.

26. Chien, R. T. and L. J. Peterson, "Optimized Computer Systems for Avionics Applications," R & D Status Report #3 on Contract No. F33615-78-C-1559, Project No. FY1175, Air Force Avionics Laboratory, WPAFB, Dayton, Ohio, May 1, 1979.

27. McClellan, J. H. and C. M. Rader, Number Theory in Digital Signal Processing, Prentice-Hall, Englewood Cliffs, N. J., 1979.

28. Jenkins, W. K., "Recent Advances in Residue Number Techniques for Recursive Digital Filtering," IEEE Trans. on Acoustics, Speech, and Signal Processing, Vol. ASSP-27, No. 1, February 1979, pp. 19-30.

29. Birtwistle, G. et al., SIMULA Begin, Auerbach, Philadelphia, 1973.

30. Various papers from Proc. International Symposium on Computer Hardware Description Languages, IEEE Press, 1975.

31. Bergland, G. D., "Fast Fourier Transform Hardware Implementation--an Overview," IEEE Trans. on Audio Electroacoustics, Vol. AU-17, June 1969, pp. 104-108.

32. Pomerleau, A., M. Fournier and H. L. Buijs, "On the Design of a Real Time Modular FFT Processor," IEEE Trans. on Circuits and Systems, Vol. CAS-23, October 1976, pp. 630-633.

33. Allen, J., "Computer Architecture for Signal Processing," Proc. of IEEE, Vol. 63, April 1975, pp. 624-633.

34. Digital Signal Computers and Processors, A. C. Salazar (ed.), IEEE Press, New York, 1977.

35. Eklundh, J. O., "A Fast Computer Method for Matrix Transposing," IEEE-TC, July 1972, pp. 801-803.

36. DAIS Processor Instruction Set, Reference Manual, Westinghouse Electric Corporation, 1976.

37. Military Standard: Airborne Computer Instruction Set Architecture, Final Working Draft, MIL-STD-1750 (USAF), October 1978.

38. Instruction Set Definition FTSC, ER78-4227, Raytheon Company, Equipment Division, April 1979.

39. Magic 362F Military, Standard Avionics Computer, S76-61, Delco Electronics, July 1976.

40. M362F-2 Computer Programming Manual, ES11442, Delco Electronics, October 1978.